像程序员一样思考

THINK LIKE A PROGRAMMER

[美] V. Anton Spraul 著　徐波 译

人民邮电出版社
北京

图书在版编目（CIP）数据

像程序员一样思考 / （美）斯保尔（Spraul, V. A.）著 ; 徐波译. -- 北京 : 人民邮电出版社, 2013.6
ISBN 978-7-115-31658-5

Ⅰ. ①像… Ⅱ. ①斯… ②徐… Ⅲ. ①程序设计－基本知识 Ⅳ. ①TP311.1

中国版本图书馆CIP数据核字(2013)第074759号

版权声明

◆ 著 [美] V.Anton Spraul
译 徐 波
责任编辑 陈冀康
责任印制 程彦红 杨林杰

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街 14 号
邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京艺辉印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：16
字数：322 千字 2013 年 6 月第 1 版
印数：1－3 500 册 2013 年 6 月北京第 1 次印刷
著作权合同登记号 图字：01-2012-7214 号

定价：49.00 元

读者服务热线：(010)67132692 印装质量热线：(010)67129223
反盗版热线：(010)67171154

内 容 提 要

编程的真正挑战不是学习一种语言的语法，而是学习创造性地解决问题，从而构建美妙的应用。本书分析了程序员解决问题的方法，并且教授你其他图书所忽略的一种能力，即如何像程序员一样思考。

全书分为 8 章。第 1 章通对几个经典的算法问题切入，概括了问题解决的基本技巧和步骤。第 2 章通过实际编写 C++代码来解决几个简单的问题，从而让读者进一步体会到问题解决的思路和应用。第 3 到 7 章是本书的主体部分，分别探讨了用数组、指针和动态内存、类、递归和代码复用来解决问题的途径和实际应用。最后，第 8 章从培养程序员思维的角度，进行了总结和概括，告诉读者如何才能像程序员一样思考。

本书选取的话题切中程序员的痛点，针对他们最容易陷入挣扎的领域展开讨论，引发思考。每章后面都给出一些编程习题，使得读者能够应用该章所讨论的概念，训练和提升问题解决的能力。

本书适合初级到中级的程序员用来提升自己的问题解决能力和应用编程技能的能力，也适合计算机相关专业的学生作为参考书阅读。

前　言

你是否在编写程序时感受到挣扎，即使觉得已经理解了自己所使用的编程语言？你是否阅读了一本编程书籍的某一章并能够顺利理解，却无法把自己所读到的东西应用于程序中？你是否能够理解自己在线所阅读的一个程序，甚至能够把每行代码所完成的任务告诉其他人，但是自己在接到一个编程任务后，却面对文本编辑器的空白屏幕大脑一片空白？

并不是只有你才这样。我从事编程教学已经超过 15 年，几乎所有的学生都在某种程度上符合上面的描述。我们称之为缺乏问题解决能力。所谓问题解决能力，就是根据问题描述编写一个原创程序来解决这个问题。并不是所有的编程任务都需要深入的问题解决能力。如果只是对一个现有的程序进行微小的修改、调试或添加测试代码，这类编程在本质上是机械的，不会对我们的创造力提出多大的挑战。但是，所有的程序总会在某个时刻需要解决问题，所有优秀的程序员都需要具备解决问题的能力。

解决问题是非常困难的。但有些人确实使它变得非常轻松，因为他们天赋异禀，就像迈克尔·乔丹这样的体育天才一样。对于这帮天才，高级的思路几乎可以毫不费力地转换为源代码。用 Java 的一个比喻来说，他们的脑子天生就像能执行 Java 代码一样，而绝大多数人只能通过虚拟机来解释代码。

没有超常天赋对于成为程序员而言并不是致命的。如果真是如此，世界上也就没有多少程序员了。但是，我看到过许多值得尊重的学习者却在挫折面前挣扎太久。最坏的情况

是，他们完全放弃了编程，认为自己永远成不了程序员，觉得只有天赋异禀的人才能成为优秀的程序员。

学习解决编程问题为什么这么困难呢？

部分原因是解决问题是一种与学习编程语言不同的活动，它使用了一组不同能力的“肌肉”。学习编程的语法、阅读程序、记忆应用程序编程接口的元素，这些都是分析性的“左脑”活动。使用以前所学会的工具和技巧编写一个原创程序却是创造性的“右脑”活动。

假设我们想拿走掉落在自家房顶上的其中一条雨槽内的一根树枝，但是自家的梯子却不够长，够不着那根树枝。我们会跑到自己家的车库，寻找某样东西来帮助我们拿走雨槽上的树枝。有没有办法使梯子加长呢？我们站在梯子上时能不能手持某样东西来抓住或拔掉树枝？或许我们可以从某个位置爬上屋顶拿走树枝。解决问题是一种创造性的活动。不管你是否相信，在设计自己的原创程序时，我们的智力活动过程与想方设法从雨槽中拿走树枝的过程非常相似，但与调试一个现有的 for 循环的过程却截然不同。

但是，绝大多数编程书籍却把重点放在语法和语义上。学习编程语言的语法和语义是极为重要的，但这只是学会用这种语言进行编程的第一个步骤。在本质上，大多数面向新手的编程书籍所讲述的是怎样阅读程序而不是怎样编写程序。把重点放在怎样编写代码上的书籍常常是有效的“烹调书”，因为它们讲述在特定情况下所使用的特定“菜谱”。这种书籍对于节省时间而言是极有价值的，但它们并不是通往学会编写原创代码的正确道路。考虑原始意义上的烹调书。尽管优秀的厨师拥有自己的烹调书，但没人依靠烹调书就能成为优秀的厨师。优秀的厨师理解配料、备菜方法和烹饪方法，并知道怎样把它们组合在一起烹制美味佳肴。优秀的厨师在烹制美味时所需要的只是工具完备的厨房。同理，优秀的程序员理解语言的语法、应用程序框架、算法和软件工程原则，并知道怎样把它们结合起来创建优秀的程序。为优秀的程序员准备一个需求规范列表，给他一个功能齐全的开发环境，他就能创造优秀的程序。

一般而言，当前的编程教育并没有在解决问题这个领域提供太多的指导。相反，如果程序员可以使用所有的编程工具并按要求编写足够的程序，最终他们将学会编写这些程序并且完成得很好。这种说法并没有错，但是“最终”可能意味着相当长的时间。从蹒跚学步到初窥门径的道路上可能充满挫折，有太多的人信心满满地开始了旅程，却倒在了半途中。

我们并不一定要饱经磨难才能获得成功，而是可以用一种系统的方式来学习怎样解决

问题，这正是本书的核心所在。我们可以学习对自己的想法进行组织的技巧、解决方案的发现过程以及对某些类型的问题所实行的策略。通过研究这些方法，我们可以释放自己的创造力。不要弄错了：编程，尤其是解决问题，是一种创造性的活动。创造性是神秘的，没人能够准确地说出创造性是怎样发挥作用的。但是，如果我们能够学会怎样作曲、能够领会别人所提出的有关写作创造能力的建议或者能够学会绘画，我们也可以学会怎样有创造力地解决编程问题。本书并不是讲述具体的做法，而是帮助读者开发属于自己的问题解决天赋，明白自己应该怎么做。本书的目的是帮助读者成为自己心目中的程序员。

我的目标是让本书的读者学会怎样用系统性的方法完成每个编程任务，并坚信自己最终能够解决一个特定的问题。当读者完成本书的学习时，我希望你能够像程序员一样思考，并坚信自己已经是个程序员。

本书内容简介

在解释了本书的必要性之后，我还需要对本书将讨论的内容和不会讨论的内容进行一些说明。

预备条件

本书假设读者已经熟悉了 C++的基本语法和语义，并且已经编写过一些程序。本书的大部分章节将预计读者已经了解了特定的 C++基础知识。这些章节将从对这些基础知识的回顾开始讨论。如果读者还在学习语言的基础知识阶段，也不必担心。关于 C++的语法有大量优秀的书籍可供参考，在学习解决问题的同时学习语法也是没有问题的。在尝试解决每章后面的习题之前，要保证自己已经学会了相关的语法。

所选择的话题

本书所覆盖的话题代表了我所看到的程序员新手最容易陷入挣扎的领域。它们还代表了初级和中级编程中许多跨领域的话题。

但是，我应该强调，这并不是一本用于解决特定问题的算法或模式的“烹调书”。尽管后面的章节讨论了怎样使用广为人知的算法或模式，但这本书并不适合作为解决特定问题的参考书，所以读者不应该只把注意力集中在直接与自己当前所面临的问题相关的章节中。

反之，读者应该从头研读全书，暂时只跳过那些由于缺乏预备知识而无法直接学习的内容。

编程风格

这里简单概述一下本书所使用的编程风格：本书并不追求高性能的编程，并不苛求产生最紧凑、最高效的代码。我为源代码例子所选择的风格是把容易理解作为首要的考虑因素。在有些情况下，我会采用多个步骤完成那些实际上只用一个步骤就可以完成的任务，其原则就是为了进行清晰的说明。

本书也会讨论编程风格的相关内容，但只是比较宽泛的问题，例如类里面应该包含什么或者不包含什么，而不会讨论细节的问题，例如代码应该怎样缩进。作为学习中的程序员，读者当然想要在自己的所有工作中采用一致的、容易阅读的风格。

习题

本书包含了一些编程习题，但本书并不是教科书，因此读者不会在书末找到这些习题的答案。这些习题为读者提供机会应用当前章节所讨论的概念。尝试完成哪些习题当然是由读者自行决定的，但是把书中所讨论的概念应用于实践是至关重要的。简单地完成全书的阅读并不能让自己的水平得到长进。记住，本书并不是告诉读者在每个解决方案中应该怎样做。在应用本书所展示的技巧时，读者能够通过拓展自己的能力来发现需要完成什么任务。而且，成功地完成这些习题也能促进本书的另一个基本目标，就是增加读者的自信心。事实上，有一种很好的方法能够知道读者是否充分地完成了一个特定问题领域的习题，那就是当读者对解决这个领域的其他问题充满信心之时。最后，编程习题应该是充满乐趣的。虽然有时候读者可能觉得辛苦而想做些其他事情，但是完成编程习题应该是一项充满回报的挑战。

读者应该把本书看成是大脑的一项障碍训练。障碍训练可以增强力量、毅力和敏捷性，为训练者培养信心。通过阅读全书并把书中所讨论的概念应用于尽可能多的习题中，可以增强信心并培养能够用于任何编程情况的问题解决技巧。在未来，当读者面临一个难题时，就会知道应该怎样克服它。

为什么用 C++

本书的编程例子是用 C++编写的。由于本书的主题是怎样解决编程问题而不是专门讨论 C++的，因此书中不会出现关于 C++特定的提示或技巧，并且本书所讨论的基本概念也适用于任何编程语言。然而，我们无法在不讨论具体程序的情况下讨论编程，因此必须选择一种特定的语言。

选择C++的原因有几方面。首先，它在许多问题领域都是非常流行的一种语言。其次，它的来源是严格的过程性语言 C，因此 C++的代码既可以采用过程性的惯用法，也可以采用面向对象的惯用法。面向对象编程现在已经极为常见，在讨论解决问题时不可能忽略它，但是许多基本的问题解决方案可以用严格的过程性编程术语来讨论，这样不仅能简化代码，而且能简化具体的讨论。第三，作为一种具有高级程序库的低层语言，C++允许我们讨论不同层次的编程。最优秀的程序员可以在需要的时候“手工编写”解决方案，并利用高级程序库和应用程序编程接口来减少开发时间。选择C++的最后一个原因是一旦学会了怎样用C++解决问题，就很容易学会用其他任何语言来解决问题。许多程序员发现在一种语言中所学会的技巧可以很轻松地应用于另一种语言，尤其是当前者是 C++语言的时候，这是由于它的惯用法跨度广泛，或者坦率地说，是由于它的难度所决定的。C++是真正的实用型语言，它并不是面向教学而设计的。虽然初看上去有些吓人，但一旦取得进展之后，就会觉得自己不再只是会一点点编程的人了，而是将成长为一名真正的程序员。

作 者 简 介

V. Anton Spraul 讲授入门级编程和计算机科学已经超过 15 年。本书凝聚了他在多年的开发经历中所提炼的经验和技巧，并在面向许多遭遇瓶颈的程序员的一对一指导中收到了良好的效果。他还是《Computer Science Made Simple》（Broadway）的作者。

译 者 简 介

徐波，浙江宁波人，熟悉 C 和 C++、Java 等语言。2002 年开始从事计算机技术图书翻译。徐波技术视野广阔，翻译文笔优美。翻译有《C 专家编程》、《C 和指针》等。

致　谢

没有一本书只通过一位作者就可以完成，在本书的写作中，我得到了很多人的帮助。

我非常感谢 No Starch 出版社的每位工作人员，尤其是 Keith Fancher 和 Alison Law，他们在本书的出版过程中担任了编辑、策划和指导工作。我还必须感谢 Bill Pollock，正是他与我签订了本书的合同，我希望他能够像我一样对本书的质量感到满意。No Starch 出版社的伙计们与我交流时表现出了从不消减的热心和诚恳。我希望有朝一日能够在私下里与他们会面，看看他们把公司网站上的卡通阿凡达装配成什么样了。

Dan Randall 作为技术编辑，他的工作非常出色。他还提出了大量技术之外的意见，帮助我在许多地方完善了手稿的质量。

在我的家庭里，生命中最重要的人 Mary Beth 和 Madeline 让我感受到爱、支持和热情，特别是为我提供了充足的写作时间。

最后，我还要感谢多年以来曾经听我讲授编程的学生们：感谢让我成为你们的老师。本书所描述的技巧和策略是在我们共同努力的过程中发展成型的。我希望我们能够让下一代程序员的学习旅程变得轻松一些。

对本书的赞誉

“作者在向初学者阐述困难概念方面显然具有广博的知识和丰富的经验。本书显示了他脚踏实地、一丝不苟却又令人愉悦的写作风格。”

—Adrian Woodhead, Slashdot

“本书所提供的习题类似于我在接受 Google 和 Facebook 的软件工程面试时所遇到的问题，因此对于打算通过面试寻找新工作的专业程序员，本书是极好的复习材料。”

—Ariane Coffin, Wired.com's GeekMom

“这是我所阅读过的收获最大的书籍之一，因为它指导我们设计一个属于自己的系统，而不是把思维固化为只能采取一种正确的方法才能达到目的。”

—Lucas Westermann, Full Circle Magazine

“如果读者能够认真研读本书，我保证它可以极大地拓展您的思维。”

—David Bolton, About.com C/C++/C#

“不管使用什么教材向新学生讲授编程和程序逻辑，我建议一定要把本书作为重要的参考书。”

—Joe Saur, The ACM's Software Engineering Notes Magazine

“V. Anton Spraul 所提供的建议简单、直观并且实用。本书的阅读是一个既轻松又极有价值的过程。”

—James Powell, Enterprise Systems

“对于所有想要培养创造性的解决问题能力的人以及已经学习了编程但觉得没有完全理解概念的人，我向他们强烈推荐本书。”

—Robert Perkins, Game Vortex

“如果我教其他人学习编程，这肯定是我将要选择的教材。”

—Stephen Chapman, Ask Felgall

目　录

第 1 章

解决问题的策略

本书的主题是怎样解决问题，但“解决问题”的确切定义是什么呢？当人们在日常谈话中提到这个术语时，所表达的意思往往与我们这里所讨论的含义截然不同。如果一辆1997年生产的本田思域轿车的排气管直冒蓝烟，怠速时震颤明显，并且油耗明显上升，说明它出现了问题。为了解决这个问题，必须具备相关的汽车知识和诊断故障的能力，并配备相应的替换部件，另外还需要一些常用的维修工具。如果把这个问题告诉朋友，可能会得到这样的建议：“嗨！你应该卖掉这辆旧本田车，然后换辆新的，问题就解决了”。但是，这位朋友的建议并不是这个问题的真正解决方案，它只是逃避这个问题的一种方法而已。

我们所说的“问题”其实包含了约束条件。约束条件就是与问题本身或者它的解决方法相关的、不能被违反的规则。以这台出了故障的思域轿车为例，其中一个约束条件就是需要修理这辆汽车，而不是购买一辆新车。其他的约束条件可能还包括修理成本、修理时限或者在修理的时候不能购买新的修理工具。

在解决程序中的问题时，也存在约束条件。常见的约束条件包括编程语言、平台（它

在 PC、iPhone 还是其他平台上运行）、性能（有些游戏程序要求图形每秒至少刷新 30 次，而商业应用程序可能要求对用户输入的响应时间设置上限）或内存需求。有时候，约束条件还涉及到可供参考的其他代码，例如程序中不能包含有开源代码，或者正好相反，它只能使用开源代码。

因此，作为程序员，我们可以把“解决问题”定义为编写一个原创程序，使它执行一组特定的任务，并满足所有预先声明的约束条件。

程序员新手常常能够非常热切地完成这个定义的第一部分，也就是编写一个程序，执行一个特定的任务。但是，他们常常受挫于这个定义的第二部分，也就是无法满足预先声明的约束条件。我把类似这样的程序（即看上去能够产生正确的结果，但是违背了一个或多个预先声明的规则）称为小林丸号（Kobayashi Maru）。如果读者对这个名称感到陌生，说明对作为极克文化试金石之一的电影《星际迷航 II：可汗怒吼》还不够熟悉。这部电影里面有一段情节，描述了星际学院中充满热情的学员们所经受的一场训练：学员们被放在一艘模拟的星舰桥上，以船长的身份接受一个不可能完成的任务。无辜的人们在一条受伤的船（小林丸号）上等待死亡。为了靠近他们，必须与克林贡战舰一战。这场战斗只可能导致船长的星舰被摧毁。这场训练是为了测试军校学生面临战火时的勇气。没有获胜的方法，所有的选择都会导致悲剧性的结果。当电影临近结束时，我们发现柯克船长更改了模拟训练装置，使它实际上可以获胜。柯克非常聪明，但他并没有解决小林丸号的困境，只是避开了它而已。

所幸的是，程序员所面临的问题通常是可以解决的，但许多程序员仍然采取了柯克船长的方法。在有些情况下，他们是不小心才这样的。（“噢！天哪，我的解决方案只有在数据项的数量不大于 100 的时候才行得通，但它必须能够处理数量不受限制的数据集，看来我必须重新加以考虑了。”）但在有些情况下，他们是有意去掉约束条件的，这是为了能够赶上老板或导师所施加的最后期限。还有一些情况，程序员只是不知道怎样满足所有的约束条件。在我所见到过的最坏的例子中，负责编程的学生花钱请人编写程序。不管出于什么动机，我们必须要想方设法避免小林丸号。

1.1 经典难题

当读者深入学习本书的时候，将会注意到尽管源代码的特定细节可能因不同的问题而异，但有些模式会一直在我们所采取的方法中出现。这是非常重要的，因为它使我们最终能够充满自信地解决任何问题，而不管我们对于当前的问题领域是否拥有丰富的经验。专

家级的问题解决者能够迅速发现类比关系，认识到一个已解决的问题和一个未解决的问题之间可供利用的相似之处。如果我们发现问题 A 的一个特性与已经解决的问题 B 的一个特性具有相似之处，就为解决问题 A 奠定了良好的基础。

在本节中，我们将讨论编程世界之外的一些经典问题，把其中的经验和教训应用于我们的编程问题。

1.1.1 狐狸、鹅和玉米

我们将要讨论的第一个经典问题是一位需要过河的农夫所面临的难题。读者以前可能已经遇到过类似性质的问题。

怎样过河

一位农夫带着一只狐狸、一只鹅和一袋玉米过河。农夫有一条划艇，只能容纳他自己加上其中一件物品。遗憾的是，狐狸和鹅都饥肠辘辘。狐狸和鹅不能单独待在一起，因为狐狸会吃了鹅。同样，也不能单独把鹅和那袋玉米放在一起，因为鹅会吃了玉米。农夫怎样才能把所有东西都送过河呢？

图 1.1 展示了这个问题的场景。如果读者以前从来没有遇到过这样的问题，可以花几分钟的时间认真研究一下怎样解决这个问题。如果以前曾经遇到过这样的问题，也可以回忆一下它的解决方案，看看自己有没有办法独立解决。

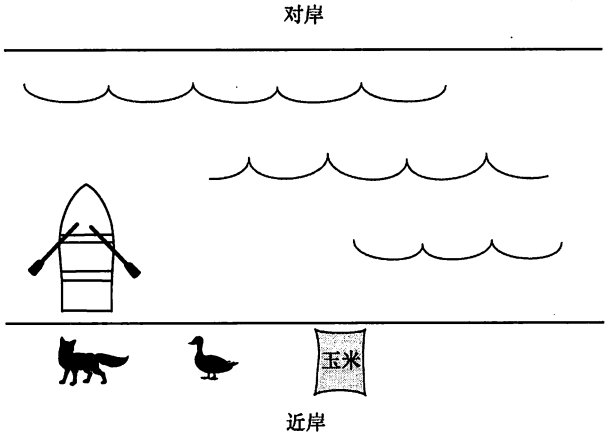

图 1.1 狐狸、鹅和一袋玉米。船一次只能载一样物品。狐狸不能单独和鹅待在同一岸边，鹅不能单独和玉米待在同一岸边

在没有任何提示的情况下，很少有人能够解出这个难题。作者也自认没有这个能耐。下面是人们解决这个问题时的通常思路。由于农夫一次只能携带一样物品，因此他需要往返多次才能把所有物品送到对岸。在第一趟过河的时候，如果农夫带走狐狸，鹅就会单独和那袋玉米待在一起，肯定会吃了玉米。类似地，如果农夫选择带走那袋玉米，狐狸就会和鹅单独呆在一起，鹅也难逃被吃的厄运。因此，农夫第一趟必须带鹅过河，从而出现如图 1.2 所示的场景。

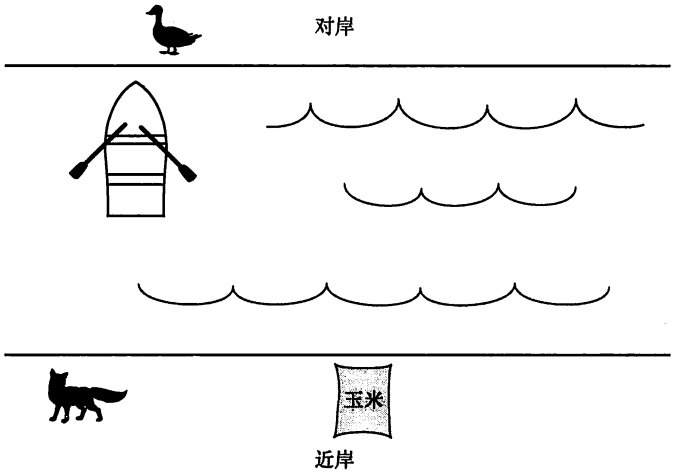

图 1.2 解决狐狸、鹅和玉米问题必然采取的第一步。但是，从这一步开始，所有后续的步骤看上去都将以失败告终

到目前为止，一切正常。但在第二趟过河时，农夫必须带走狐狸或玉米。但是，不管这次农夫带走什么，当农夫从对岸返回取最后一件物品时，第二趟所带去的物品必须在对岸与鹅单独待在一起。这意味着，要么是狐狸和鹅独处，要么鹅和玉米独处。由于这两种情况都是无法接受的，因此问题看上去似乎是无法解决的。

如果读者以前曾经看到过这个问题，很可能还记得解决方案的关键要素。如前所述，农夫在第一趟时必须带走鹅。在第二趟时，我们假设农夫带走了狐狸。但是，他并不是让狐狸和鹅一起待在对岸，而是在返回时把鹅带回近岸。然后，农夫把玉米带到对岸，将狐狸和玉米放在一起后返回。在第四趟过河时，农夫最后把鹅带走。图 1.3 展示了完整的解决方案。

这个问题很难，因此大多数人从来不会想到把物品再从对岸带回近岸。有些人甚至觉得这个问题是不公平的，表示：“你并没有说我可以把物品带回来！”确实如此，但是在描述问题的时候，题目也没有说禁止从对岸把物品带回来。

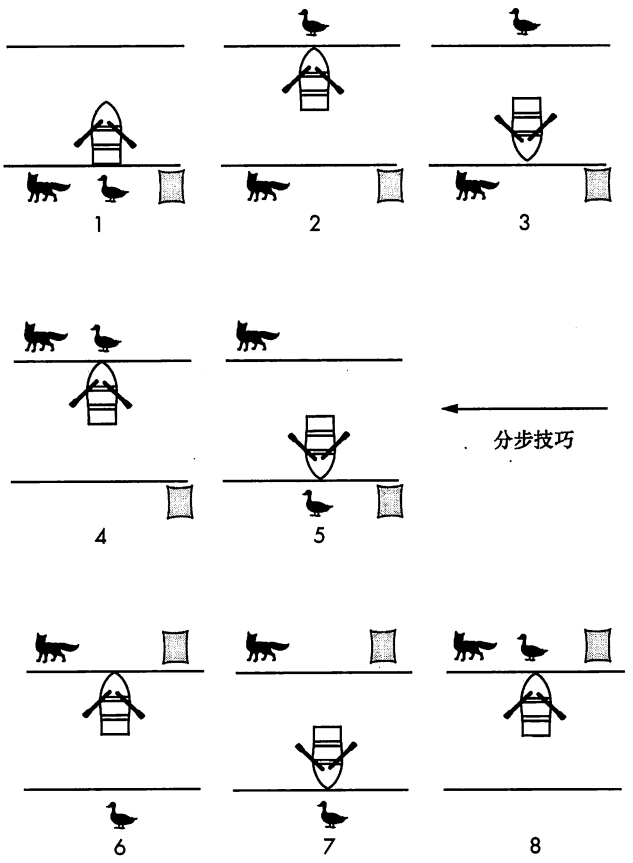

图 1.3 狐狸、鹅和玉米难题的一步步解决方案

可以想象一下，如果事先明确说明可以把物品从对岸带回来，这个问题就会相当容易解决：

农夫有一条划艇，可以来回搬运物品，但小艇每次只能容纳农夫自身再加上他的三件物品之一。

有了这个说明之后，很多人都能解决这个问题了。这说明了解决问题的一个重要原则：如果没有意识到所有可以采取的动作，很可能无法解决问题。我们可以把这些动作称为操作。通过列举所有可能的操作，可以解决许多问题。我们只要测试这些操作的每种组合，直到发现可行的方案。概括地说，就是用更加形式化的术语重新陈述一个问题，常常可以发现此前被我们所忽略的解决方案。

我们先把这个问题的解决方案抛之脑后，然后用更形式化的方式陈述这个特定的难题。

首先，我们列出了约束条件。这个问题的关键约束条件如下。

1．农夫在船中一次只能放一件物品。

2．狐狸和鹅不能单独放在同一岸边。

3．鹅和玉米不能单独放在同一岸边。

这个问题是说明约束条件重要性的一个很好的例子。如果我们去除所有的约束条件，这个问题就毫无难度。如果我们去除第一个约束条件，可以简单地一次携带全部 3 件物品过河。即使一次只能在船上放两件物品，也可以让狐狸和玉米先过河，然后再带鹅过河。如果我们去掉第二个约束条件（但保留另两个约束条件），就必须小心谨慎了。首先要带鹅过河，然后带狐狸过河，最后带玉米过河。因此，如果我们忘了或忽略了任何一个约束条件，就相当于遇到了小林丸号这样的情况。

接着，我们列出所有的操作。陈述这个问题的操作可以采用多种不同的方式。我们可以创建一个特定的列表，列出所有可以采取的操作。

1．操作：把狐狸带到河的对岸。

2．操作：把鹅带到河的对岸。

3．操作：把玉米带到河的对岸。

但是必须谨记，以形式化的方式重新陈述问题的目标是为了能够更好地发现解决方案。除非我们已经解决了问题并发现了“被隐藏的”可能操作（也就是把鹅带回到近岸），否则我们是无法通过上面所列出的这些操作方式发现这个操作的。因此，我们应该设法列出更基本（或参数化）的操作。

1．操作：把船从岸的一边划到另一边。

2．操作：如果船为空，从岸上装载一件物品。

3．操作：如果船不为空，把物品卸到岸上。

通过用最基本的术语来考虑问题，上面的操作列表可以让我们轻松地解决这个问题，就不会把“把鹅从对岸带回近岸”看成是什么奇思妙想了。如果我们枚举所有可能的移动序列，当每个序列违反了其中一个约束条件或重复了此前已经看到过的一个配置时就终止该序列，最终能够得到图 1.3 所描述的序列并解决这个问题。通过形式化地重新陈述这个问

题的约束条件和操作，就成功地避开了它内在的困难性。

经验和教训

我们可以从狐狸、鹅和玉米的问题中学到什么呢？

用更形式化的方式重新陈述问题是一种非常出色的技巧，可以让我们拥有对问题更好的洞察力。许多程序员设法与其他程序员一起讨论问题，并不仅仅因为对方可能已经有了答案，而是因为清晰地陈述问题常常会激发有用的新思路。重新陈述问题就相当于与其他程序员讨论问题，只不过现在是一人分饰两角。

更深远的意义在于，认识到思考问题很可能与思考解决方案具有相同的工作效率，甚至更胜一筹。在许多情况下，通往解决方案的正确道路本身就是解决方案。

1.1.2 瓷砖滑块问题

瓷砖滑块问题存在许多不同的规格，正如我们稍后所看到的那样，它提供了一种特定的解决机制。下面是对 3×3 版本的瓷砖滑块问题的描述。

滑动 8 块

一个 3×3 的网格中放了 8 块瓷砖（编号从 1～8），剩下一格为空。一开始，网格的配置很杂乱。瓷砖可以滑动到邻近的空格中，使它的原先位置为空。这个问题的目标是在网格中滑动瓷砖，使它们从左上角开始在网格中有序地排列。

图 1.4 展示了这个问题的目标。如果读者之前从来没有遇到过这样的问题，可以花些时间考虑怎么解决。网络上可以找到大量的滑动瓷砖问题模拟程序，但最好还是用扑克牌或索引卡在桌上试验。图 1.5 显示了推荐的初始配置。

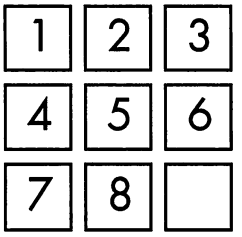

图 1.4 8 块瓷砖版本的瓷砖滑块问题的目标配置，空格表示邻近的瓷砖可以移动到的空间

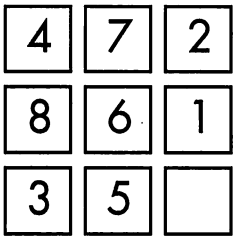

图 1.5 瓷砖滑块问题的一个特定初始配置

这个问题与前面所讨论的农夫与狐狸、鹅和玉米的问题截然不同。过河问题的难度在于可能会忽略其中一种可行的操作。在这个问题中，并不会发生这样的情况。对于任何特定的配置，空格周围最多有 4 块瓷砖，它们都可以移动到这个空格中。这样的描述已经枚举了所有可能的操作。

这个问题的难度在于解决方案所需的漫长的操作环节。一系列的滑动操作可能把某些瓷砖滑动到它们的目标位置，同时又把其他瓷砖移出正确位置，或者可能把某些瓷砖滑动到靠近目标位置，同时又把其他瓷砖滑动到远离目标位置。由于这个原因，很难认定任何特定的操作是否朝着最终的目标迈近了一步。因为没有办法衡量进度，所以很难形成一种策略。很多人通过随机的滑动来解决这个问题，希望恰好能够滑动到最终的目标配置。

但是，瓷砖滑块问题还是存在策略的。为了演示其中的一种方法，我们可以考虑一个更小的网络，它是长方形的，而不是正方形的。

滑动 5 块

有一个 2×3 的网格里放了 5 块瓷砖（编号从 4～8），另外剩下 1 个空格。一开始，这些瓷砖排列得很混乱。瓷砖可以滑动到邻近的空格中，使自己原来的位置成为空格。这个问题的目标是使这个网格中的瓷砖排列有序，4 号瓷砖出现在网格的左上角，接下来依次类推。

读者可能注意到这几块瓷砖是以 4 到 8 编号的，而不是从 1 到 5。读者很快就能知道这样安排的原因。

尽管它是与 8 块的瓷砖滑块相同的基本问题，但只有 5 块瓷砖显然要简单得多。尝试完成如图 1.6 所示的配置。

如果试上几分钟，很可能会找到一种解决方案。从较小数量的瓷砖滑块问题出发，我

开发了一个特定的技巧。这个技巧加上我们稍后将要进行的讨论，可以用来解决所有的瓷砖滑块问题。

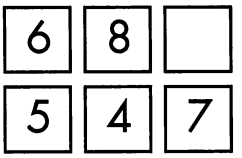

图 1.6 2×3 版本的瓷砖滑块问题的一个特定起始配置

我把这种技巧称为“串列”，它基于对一组瓷砖位置所形成环路的观察。这些位置加上空格就构成了一个瓷砖串列，可以在这个环路中旋转，同时保持这些瓷砖的相对顺序。图 1.7 演示了由 4 个位置所组成的最小可能串列。在一开始的配置中，1 可以滑动到空格中，2 可以滑动到 1 移走后所留下的空格中，最后 3 可以滑动到 2 移走后所留下的空格中。这样，空格就与 1 相邻，使这个串列可以继续旋转。因此，这些瓷砖可以在这条串列路径中有效地旋转到任何位置。

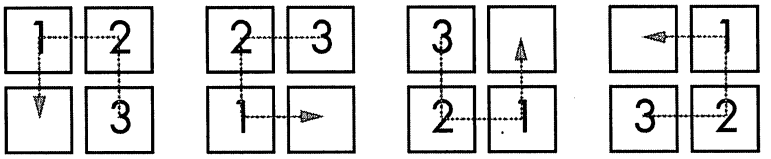

图 1.7 “串列”，与空格相邻的瓷砖开始形成了一条瓷砖路径，在游戏过程中可以像一列火车一样滑动

我们可以使用串列移动一系列的瓷砖，同时保持它们之间的相对关系。现在我们回到前面的 2×3 的网格配置。尽管这个网格中没有任何一个瓷砖位于它最终的正确位置，但有些瓷砖却靠近它们在最终配置中需要靠近的瓷砖。例如，在最终的配置中，4 将出现在 7 的上面，而这两块瓷砖当前是相邻的。如图 1.8 所示，我们可以用一个包含 6 个位置的串列把 4 和 7 移动到它们最终正确的位置。当我们完成这个操作时，剩余的瓷砖几乎都处于正确的位置，只需要再移动一下 8 就可以了。

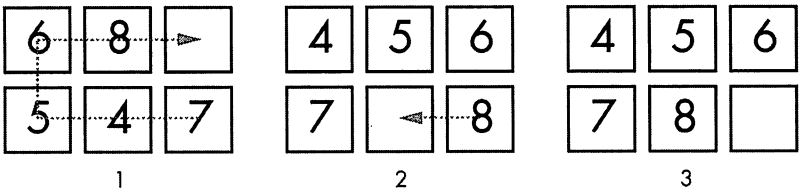

图 1.8 从配置 1 出发，经过 2 次沿规划的“串列”旋转之后来到配置 2，然后只要滑动 1 次就可以产生最终的配置 3

这种技巧是怎么解决所有瓷砖滑块问题的呢？考虑最初的 3×3 配置。我们可以用包含 6 个位置的串列移动相邻的 1 和 2，使 2 和 3 相邻，如图 1.9 所示。

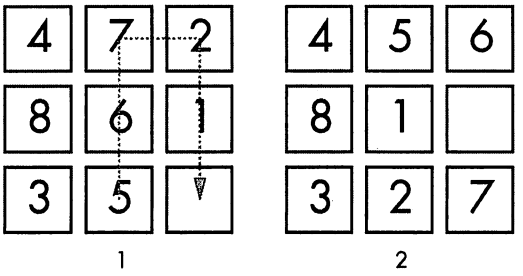

图 1.9 从配置 1 出发，瓷砖沿规定的“串列”旋转到达配置 2

这样 1、2 和 3 就相邻了。在 8 个位置的串列中，我们就可以旋转 1、2 和 3 了，使它们到达最终的正确位置，如图 1.10 所示。

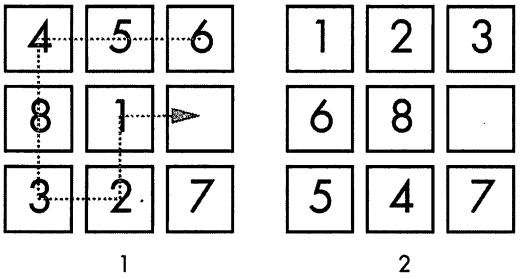

图 1.10 从配置 1 出发，瓷砖经过旋转之后到达配置 2，这样瓷砖 1、2 和 3 位于正确的最终位置

注意瓷砖 4～8 的位置。这些瓷砖的配置正好与前面的 2×3 网格的例子相同。这是一个至关重要的观察。把瓷砖 1～3 放在正确的位置之后，我们就可以把剩余的瓷砖看成是一个更小、更容易解决的独立问题。注意，为了使这种方法可行，必须要解决一整行或一整列的瓷砖。如果我们把瓷砖 1 和 2 放在正确的位置，但 3 仍然置于别处，那样就无法在不移动左上角两块已经就绪的瓷砖的情况下，把任何瓷砖移动到右上角位置。

这个技巧也可以用于解决规模更大的瓷砖滑块问题。常见的最大尺寸是 15 块瓷砖，也就是 4×4 的网格。这样的问题也可以用分解法来解决：首先把瓷砖 1～4 移动到正确的位置，这样就只剩下一个 3×4 的网格，然后再完成最左列的瓷砖，这样就只剩下一个 3×3 的网格。此时，就只剩下 8 块瓷砖需要移动了。

经验和教训

我们可以从瓷砖滑块问题中学到什么呢？

由于瓷砖移动的次数相当之多，因此无法在初始配置时就规划一个完整的解决方案。但是，无法规划完整的解决方案并不意味着就无法采取策略或技巧系统性地解决问题。在解决编程问题时，有时候会出现无法看到通向解决方案的清晰道路的情况，但这绝不能成为跳过计划和采用系统性方法的借口。更好的办法是采用一种策略，而不是通过简单地反复尝试和失败来解决问题。

我是通过对较小的问题进行研究时发现这种“串列”技巧的。在编程中，我常常会使用这种类似的技巧。在面临一个复杂的问题时，我常常会对这个问题的削减版本进行试验。这些试验常常能够产生有价值的思路。

另一个经验是问题的细分通常并不是非常明显的解决之道。由于移动一块瓷砖不仅影响这块瓷砖本身，还会影响接下来可能发生的移动，人们可能觉得瓷砖滑块问题必须在一个步骤中完成，而不能分阶段解决。因此，花时间研究怎样对问题进行细分通常是非常合算的。即使无法找到一种清晰的细分，仍然有助于增强对这个问题的理解，可以促进这个问题的解决。在解决问题时，头脑里已经拥有一个特定的目标总比随机的尝试要好得多，无论最终是否能够实现这个目标。

1.1.3 数独

数独游戏作为一种在报纸和杂志上出现的益智游戏，其流行程度已经达到了惊人的地步，并且已经发展成一种基于网络和手机的游戏。数独拥有许多不同的版本，但这里只简单讨论传统的版本。

完成数独方块

一个 9×9 的网格，其中部分方格填有数字（1～9），玩家必须填满剩余的空格，并满足下面这个约束条件：对于每一行和每一列，每个数字恰好只出现 1 次。而且，对于粗框内的每个 3×3 区域，每个数字也恰好只出现 1 次。

如果读者以前曾经玩过这个游戏，很可能已经知道应该采用什么策略在最短的时间内完成一个空格。我们首先观察如图 1.11 所示的方块，关注最关键的起始策略。

数独游戏的难度差异极大，它们的难度取决于需要填充的空格数量。按照这种衡量方法，这是一个相当容易的问题，因为已经有 36 个空格已经填好，只需要再填写 45 个空格就可以完成任务了。问题在于，我们应该首先填充哪个空格呢？

	9	1		6		7		
				8	2		3	9
5		3				2		
			9	1	3		6	2
		2	4		6	8		
1	4		8	2	5			
		9				5		7
6	7		1	5				
		5		4		6	9	

图 1.11 一个很简单的数独方块问题

记住，这个问题包含了约束条件。在每一行、每一列以及粗框内的每个 3×3 区域中，9 个数字必须都正好出现 1 次。这些规则决定了我们应该从哪里着手。最中间的 3×3 区域的 9 个方格已经有 8 个填好了数字。因此，正中心的那个空格只可能填入这个 3×3 区域的其他方格没有出现的那个数字。我们应该从这个空格开始解决这个问题。这个区域所缺少的数字是 7，因此我们在最中间的那个空格中填入 7。

填好了这个数字之后，注意最中间那列的 9 个值已经有 7 个已经填好，只剩下 2 个空格，必须填上这一列所缺少的两个值：3 和 9。与这个列有关的约束条件允许把这两个值放在任意一个位置，但是注意 3 已经在第三行出现过，9 已经在第七行出现过。因此，我们应该把 9 填在中间那列的第三行，把 3 填在中间那列的第七行。图 1.12 对这些步骤进行了概括。

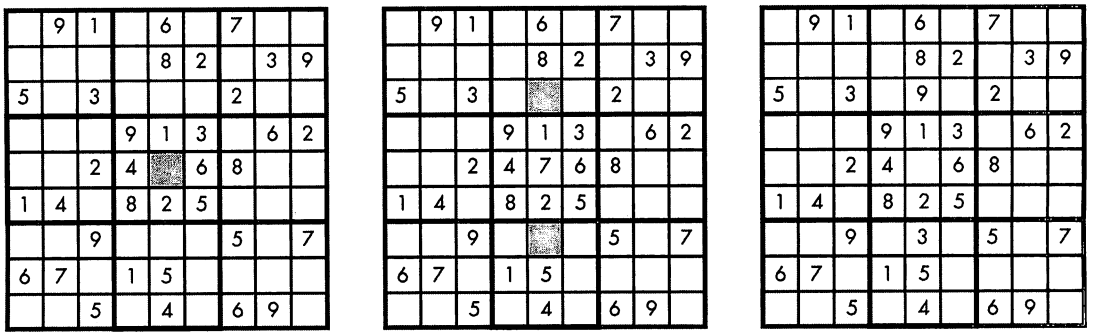

图 1.12 解决示例数独问题的开始步骤

我们不打算完成整个方块的填写，但前面这几个步骤已经说明了重要的一点，就是搜索那些可能出现的值最少的空格。在最理想的情况，就是只剩下一个空格。

经验和教训

我们在数独问题中所学到的主要经验就是应该寻找问题约束性最强的部分。虽然约束条件往往使问题难以着手（还记得狐狸、鹅和玉米问题吗），但它们也可以简化思路，因为它们消除了很多选择。

尽管我们不会在本书中详细讨论人工智能，但还是要简单地提一下，在人工智能中解决某些类型的问题时有一个称为“最大约束变量”的规则。它表示在一个问题中，当我们向一些不同的变量赋一些不同的值来满足约束条件时，应该从约束性最强的变量开始。换用更通俗的说法，就是那些可能采用的值具有最少的变量。

下面是这种思维方式的一个例子。假设有一组工友计划一起吃午饭，并且想要找一家每个人都喜欢的餐厅。问题在于，每个工人对于整个小组的决策都会施加某种程度的影响。例如，Pam 是个素食主义者，Todd 不喜欢中国菜等。如果目标是最大限度地减少寻找餐厅的时间，首先应该询问对餐厅最挑剔的那个工人。例如，如果 Bob 对很多食物都过敏，首先列出他可以进食的餐厅列表是非常合理的。像 Todd 不喜欢中国菜这样的癖好应该放在最后考虑，因为这个困难是很容易克服的。

这个技巧往往也可以用于编程问题。如果问题的某个部分具有很强的约束条件，很可能应该从这一部分开始着手，这样就不必担心把时间花在将来可能会返工的任务上了。一个相关的推论是：应该从最显而易见的那部分任务开始着手。如果可以解决这个部分的问题，就可以在此基础上继续执行其他可以完成的任务。通过审视自己的代码，可能会激发自己的想象力，从而解决剩余部分的问题。

1.1.4 Quarrasi 锁

对于上面这几个问题，读者以前可能也看到过。但是对于本章的最后一个问题，除非以前曾经阅读过本书，否则绝不可能见过，因为这是我自己“发明”的。请认真阅读，因为这个问题的描述稍微有点复杂。

打开外星锁

一种敌对的外星生物 Quarrasi 登陆到地球上，你在战斗中被它们抓

获并关押在飞船里。你设法打倒了看守，尽管它们体形庞大并且长有触角。但是，为了逃离飞船（仍然在地面上），必须打开巨大的舱门。开门的指令非常奇怪，它是用英语的形式显示的，但非常难弄。为了打开舱门，必须沿着轨道滑动 3 个条状的 Kratzz，从右侧的接收器滑动到位于门尽头的左侧接收器，距离大约 3m。

这个任务相当简单，但是必须避免触发警报。警报的工作原理如下：每个 Kratzz 就是一个或多个星形的水晶力量宝石，称为 Quinicrys。每个接收器具有 4 个传感器，如果一个纵列中 Quinicrys 的数量为偶，它们就会被点亮。如果被点亮的传感器的数量正好为 1，就会发出警报。好消息是每个警报都配备了一个抑制器，只要按下这个按钮，就可以防止警报发出声音。如果可以同时按下所有的抑制器，问题就非常简单了，但是没有办法做到这一点，因为人类的胳膊过于短小，不比长长的 Quarassi 触角。

根据上面的描述，怎么才能在不触发任一警报的前提下滑动 Kratzz 打开舱门呢？

图 1.13 展示了初始配置，3 个 Kratzz 都位于右侧的接收器。为了清晰起见，图 1.14 展示了一种不好的思路：把最上面那个 Kratzz 滑动到左侧接收器会导致右侧接收器处于报警状态。你可能想到用抑制器来避免报警，但是要记住，你刚刚把最上面的 Kratzz 移动到左侧接收器，够不着相距 3m 的右侧接收器上的抑制器。

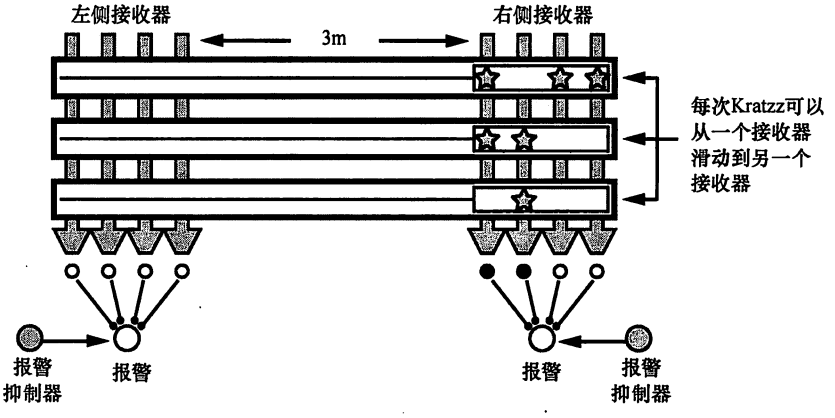

图 1.13 Quarrasi 锁问题的初始配置。必须滑动当前位于右侧接收器的 3 个 Kratzz 条，在不触发任何一个警报的情况下把它们滑动到左侧的接收器。当偶数个星形的 Quinicrys 出现在上面的纵列时，就会点亮一个传感器，如果正好有一个被连接的传感器被点亮，就会触发警报。抑制器可以防止警报发声，但你只能控制自己所站那一侧的抑制器

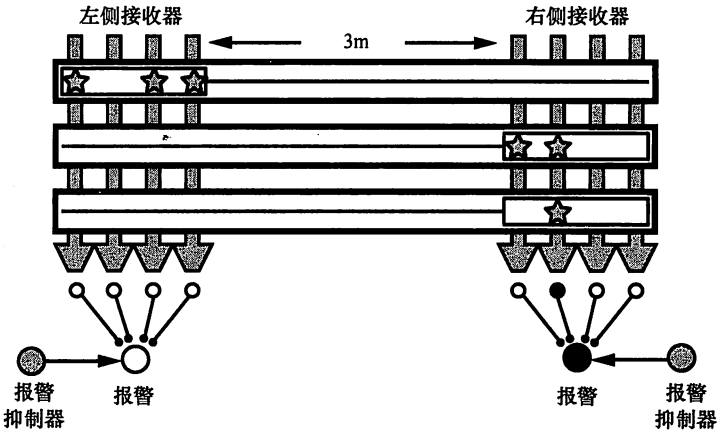

图 1.14 处于报警状态的 Quarrasi 锁。你刚刚把最上面的 Kratzz 滑动到左侧的接收器，因此够不到右侧的接收器。右侧警报的第 2 个传感器被点亮，因为出现在那个纵列的 Quinicrys 数量为偶，现在正好是一个传感器被点亮，所以就会触发报警

在继续尝试之前，先花点时间研究这个问题，设法确定一个解决方案。这取决于看问题的着眼点，此问题并没有看上去那么难。认真地说，要在尝试之前先对它进行思考！

考虑好了吗？现在是不是能够想出一个解决方案？

为了回答这个问题，可以选择两条可能的路径。第一条路径就是不断尝试，不过它是错误的做法：尝试用各种方式移动这几个 Kratzz，一旦达到警报状态时就返回到前一步骤，直到最终通过一系列的移动，成功地打开锁。

第二条路径是认识到这个问题实际上是个机关。你从前可能没看到过这种机关，它实际上就是披着伪装外衣的狐狸、鹅和玉米问题。尽管警报的规则是以通用的方式描述的，但是与这种特殊的锁有关的组合却是有限的。如果只考虑 3 个 Kratzz，我们只需要知道接收器上的哪些 Kratzz 组合是可以接受的。如果我们把这 3 个 Kratzz 分别命名为 top、middle 和 bottom，那么会触发报警的组合是“top 和 middle”以及“middle 和 bottom”。

如果我们把 top 重新命名为狐狸，把 middle 重新命名为鹅，把 bottom 重新命名为玉米，这样所有的麻烦组合都与狐狸、鹅和玉米问题一样了，也就是“狐狸和鹅”和“鹅和玉米”。

因此，这个问题的解决方式与狐狸、鹅和玉米问题相同。我们把 middle（鹅）滑动到左侧的接收器，再把 top（狐狸）滑动到左侧，当我们移动 top（狐狸）时摁住左侧警报的抑制器；接着，我们把 middle（鹅）滑动回右侧的接收器；然后，我们把 bottom（玉米）滑动到左侧；最后，我们把 middle（鹅）再次滑动到左侧，这样就打开了锁。

经验和教训

这个问题向我们提供的主要经验就是认识到类比的重要性。我们可以看到 Quarrasi 锁问题实际上与狐狸、鹅和玉米问题非常相似。如果我们早早就发现了这种类比，就可以根据狐狸、鹅和玉米问题直接想到解决办法，而不需要从头创建一个新的解决方案，从而大大节省了精力。在解决问题时，大多数类比并不是这样直接，但它们的出现频率却非常高。

如果读者难以发现这个问题和狐狸、鹅和玉米问题之间的联系，这只是因为我故意增加了一些多余的细节。与 Quarrasi 锁问题相关的背景对于解决这个问题而言是无关紧要的，那些外星术语也是如此，它们唯一的作用就是让读者感觉生疏。而且，警报的奇/偶机制使这个问题看上去比实际更为复杂。如果观察 Quinicrys 的实际位置，就可以看到顶部的 Kratzz 和底部的 Kratzz 正好相对，因此它们并不会与警报系统交互。但是，中间的 Kratzz 却与另两个发生交互。

同样，如果没有发现类比，也不必担心。对它们有了足够的警觉之后，就很容易认识到它们。

1.2 基本的问题解决技巧

我们所讨论的这些例子说明了在解决问题时将要采用的许多关键技巧。在本书的剩余部分，我们将观察特定的编程问题，并想办法怎样解决它们。但是，我们首先需要了解一组基本的技巧和原则。有些问题领域需要使用特定的技巧，但下面这些规则几乎适用于所有的场合。如果把它们作为自己的问题解决方法的常规武器，就能想出办法解决任何特定问题。

1.2.1 总是要制订计划

这也许是最重要的规则。我们事先必须要制订计划，而不是直接进行漫无方向的尝试。

根据这个观点，我们应该理解事先制订计划总是可能的。如果在头脑里还不知道怎么解决问题，就不可以制订计划编写代码实现一个解决方案。这个任务是在以后完成的。但是，即使是在开始阶段，还是应该为怎样寻找解决方案制订一个计划。

平心而论，这样的计划可能需要在其他阶段进行修改，或者必须抛弃原先的计划并制

订一个新计划。既然如此，为什么这个规则仍然非常重要呢？艾森豪威尔将军有句名言：“我总是发现计划没什么用处，但计划仍然是必不可少的。”他的意思是战争是极为混乱的，事先预测将要发生的每件事情并为每种结果准备预定的方案是不可能的。从这个意义上说，计划在战场上是没有用处的（普鲁士军事领袖赫尔默特·冯·毛奇曾有名言“一旦与敌人交上火，所有计划都不再有效”）。但是，如果没有计划和组织，任何军队都不可能取得胜利。通过精心制订计划，将军能够了解敌军的战斗力、部队中不同部门的配合方式等重要信息。

同样，我们在解决一个问题时必须要制订一个计划。也许这个计划一旦与敌人交上火就不再有效，也许我们在源代码编辑器中开始输入代码时就会抛弃这个计划，但我们还是必须制订计划。

如果没有计划，我们只能简单地希望“摔出好运气”，相当于在键盘上随意输入却产生了莎士比亚的伟大作品。“摔出好运气”是极为罕见的，而且就算如此可能仍然需要计划。很多人听说过青霉素的发现过程：一位名叫亚历山大·弗莱明的研究人员在一个晚上忘了盖上一个培养皿，第二天早上他发现霉菌抑制了培养皿上细菌的生长。但是，弗莱明并不是干坐在那里等待摔出好运气，他已经按照一种精心可控的方式进行了试验，因此能够认识到他在培养皿上所看到的事实的重要性。（如果我发现前一天晚上所放置的某样东西上长了霉菌，肯定不会对科学产生重大的贡献。）

计划还允许我们设置中期目标并实现它们。如果没有计划，我们就只有一个目标：解决整个问题。在解决了整个问题之前，我们不会感觉自己实现了什么目标。我们很可能有过这样的经验，许多程序在接近完成前没有实现任何实用的功能。因此，只朝着主要目标努力会不可避免地遭受挫折，因为在工作结束之前，不会有正面的推动力激励自己。反之，如果我们创建了一个具有一系列分阶段目标的计划，虽然看上去与主要的问题不是非常密切，仍然可以朝着解决方案取得可衡量的进展，并感觉自己的时间被合理地使用了。在每个工作阶段完成时，我们就可以检查计划所制订的各个事项，更有信心找到解决方案，而不是被日益加重的挫折感所笼罩。

1.2.2 重新陈述问题

狐狸、鹅和玉米的问题鲜活地证明了重新陈述问题可能会产生非常有价值的结果。在某些情况下，一个看上去非常困难的问题如果用一种不同的方式或术语进行阐述，就会变得非常容易。重新陈述问题就像是在登山之前寻找一个不同的出发地点。在登山之前，为什么不从每个角度进行观察，确定最容易攀登的路线呢？

重新陈述问题有时候可以向我们展示此前没有想到的目标。我曾经看过一个故事，一位祖母一边编着毛衣一边照看孙女。为了在织毛衣时不受干扰，祖母把婴儿放在她旁边的一个可移动游戏围栏中，但她的孙女不喜欢待在里面，不停地哭泣。祖母尝试了把所有各种类型的玩具放在围栏里都没有收到效果。最终，她才意识到把孙女放在围栏里只是实现目标的一种方法，她的真正目标是能够安静地织毛衣。解决方案是：让孙女在地毯上自由自在地玩耍，祖母则坐在围栏里面专心织毛衣。重新陈述可以成为一种威力强大的技巧，但很多程序员会忽略它，因为它并不直接与编写代码有关，甚至和解决方案的设计也没什么关联。这是制订计划之所以重要的另一个原因。如果没有计划，我们的目标就是可运行的代码，而重新陈述问题则是将时间用在与编写代码没有关系的地方。有了计划之后，我们就可以把“形式化地重新陈述问题”作为自己的第一个步骤，完成重新陈述就意味着取得了进展。

即使重新陈述问题并没有直接让我们获得新思路，它仍然可能在其他方面提供帮助。例如，如果我们碰到了一个问题（由上级或指导老师所“指派”），我们可以把问题重新陈述给指派这个任务的人，以确认自己的理解无误。另外，重新陈述问题对于使用其他常用的技巧也可能是一个必要的先决步骤，例如削减或划分问题。

总而言之，重新陈述问题可以转换整个问题领域。我在第 6 章的递归解决方案中所使用的技巧就是一种重新陈述递归问题的方法，使我可以像处理迭代问题一样处理它们。

1.2.3 划分问题

找到一种方式把一个问题的解决方法划分为几个步骤或几个阶段，可以使问题更容易解决。如果我们把一个问题划分为两个片段，可以认为每个片段的难度相当于原先整个问题的一半，但通常还要容易得多。

如果读者了解常用的排序算法，对下面这个类比应该是非常熟悉了。假设我们需要把 100 个文件按照字母顺序放在一个箱子里，并且我们采用的基本字母排序方法是一种很有效的被称为插入排序的方法：随机取其中一个文件，把它放在箱子里，然后把第 2 个文件放在箱子中相对于第 1 个文件正确的位置，接着继续这个过程，总是把新文件放在箱子里相对于其他文件正确的位置。因此，在任一特定时刻，箱子中的文件就是按字母顺序排列的。假设有人一开始粗略地把这些文件分成数量大致相等的 4 组：A～F、G～M、N～S 和 T～Z，然后告诉我们分别对各组文件按字母顺序排列，最后依次把各组文件放在箱子里。

如果每组大约包括了 25 个文件，人们可能觉得分别对 4 组 25 个文件按字母排序所需

要的工作量和对单组 100 个文件按字母排序的工作量是一样的。实际上，前者的工作量要小得多，因为插入一个文件所需要的工作量是随着已经排好序的文件数量的增加而增长的。我们必须检查箱子中的每个文件才能知道这个新文件应该放在哪个位置。（如果对此感到怀疑，可以考虑一个更极端的版本，比较一下对 50 组 2 个文件进行排序的方法，很可能不到一分钟就能完成全部 100 个文件的排序。）

同样，对问题进行划分常常可以使问题的难度大幅度降低。把各个编程技巧组合在一起使用要比单独使用每个技巧困难得多。例如，如果一段代码在一个 while 循环内部包含了一系列的 if 语句，而这个循环本身又位于一个 for 循环的内部，这样的代码是很难编写和理解的。相对而言，按照顺序编写这些相同的控制语句则要容易编写和理解很多。

我们将在后面的章节中讨论划分问题的具体方法。但是，我们终归应该意识到这种可能性。记住，像瓷砖滑块这样的问题常常隐藏着潜在的划分方法。有时候，寻找问题的划分方法就是削减问题的方法，正如我们稍后将要讨论的一样。

1.2.4 从自己所知的开始

作家在开始创作的时候，常常会收到这样的建议“写自己知道的东西”。这并不意味着作家只能描写他们在日常生活中直接观察到的人和事。如果这样，我们就无法看到奇幻小说、历史小说以及其他许多流派的小说了。但是，这个建议的意思是，作家所写的东西距离他自己的经历越远，写作的难度也就越大。

同样，在编程的时候，我们应该尽量从自己知道的部分开始着手。例如，一旦我们把问题划分为几个片段，应该寻找自己已经知道怎样编写代码的片段。完成了解决方案的一部分之后，可能会激发完成剩余工作的灵感。另外，正如我们可能预料到的那样，问题解决领域的一个常见主题就是取得实际的进展以构建自己的信心，相信自己能够最终完成整个任务。通过从自己所知的领域开始着手，我们就能够构建获得成功的信心和动力。

“从自己所知的开始”这句格言还适用于尚未对问题进行划分的时候。想象一下，有人创建了一个包含每个编程技巧的完整列表：编写一个 C++类、对一个数值列表进行排序、寻找一个链表的最大值等。作为程序员，在开发过程的每一刻，这个列表中可能有许多任务是我们所掌握的，有些是需要花费一些心思的，还有一些则是现在还不知道怎么解决的。一个特定的问题用自己已经掌握的技巧可能是完全可以解决的，也可能是无法解决的。但是，在寻找其他方法之前，我们应该用已经掌握的技巧对问题进行完整的研究。如果我们把编程技巧看成是工具，把编程问题看成是家庭维护项目，首先应该用手头上已经有的工

具进行修理，然后再考虑到五金店购买新工具。

这个技巧遵循了我们已经讨论的原则。它遵循了一个计划，指挥我们的工作。当我们用自己所掌握的技巧对一个问题进行研究时，可以更好地理解这个问题本身以及它的最终目标。

1.2.5 削减问题

当我们面临一个无法解决的问题时，通过这种技巧，可以削减问题的范围。我们可以添加或取消约束条件，产生一个自己知道如何解决的问题。在后面的章节中，我们将通过实际例子讨论这种技巧，不过下面先通过一个基本的例子阐述它的概念。假设一个三维空间中有一系列的坐标，我们的任务是找到空间距离最近的那对坐标。如果我们并不能立即知道怎样解决这个问题，可以采用一些不同的方式削减这个问题以寻求解决方案。例如，如果这些坐标是在一个二维空间而不是三维空间中如何是好？如果削减成这样还是无法解决，那么把二维空间再缩减为一条直线，这些坐标就成了各个不同的数（是否可以用 C++ 的 double 类型表示），这样不是就可以解决了吗？现在，这个问题在本质上就成了在一个数的列表中寻找两个具有最小绝对差的数。

或者，我们在削减问题时仍然让这些坐标位于三维空间中，但只处理 3 个值，而不是任意数量的坐标系列。因此，现在问题就不再是寻找一种算法，计算两个任何坐标之间的距离，而是变成了坐标 A 与坐标 B 比较、坐标 B 与坐标 C 比较，然后是坐标 A 与坐标 C 比较。

这两种削减方法通过不同的方式简化了原先的问题。第一种削减方法消除了计算三维点之间距离的需要。也许我们还不知道该怎样计算空间距离，但在知道怎么做之前，仍然可以取得一些进展。反之，第二种削减方法仍然要求我们计算三维点之间的距离，但是消除了在任意数量的三维点之间寻找最小值的问题。

当然，为了解决最初的问题，我们最终需要组合这两种削减方案所涉及的技巧。即使如此，削减问题允许我们对一个更简单的问题进行操作，即使我们无法找到一种方法把问题划分为几个阶段。在实际效果上，它相当于故意同时、又是临时的小林丸号。尽管我们知道并不是对整个问题进行处理，但是经过削减的问题与原先的问题仍然具有相当多的共性，足以让我们向最终的解决方案又迈进一步。许多时候，程序员发现他们具备了解决问题所需要的所有单独技巧，通过为问题的每个单独片段编写代码，他们可以想到怎样把各个不同的代码片段组合为一个统一的整体。

削减问题还允许我们准确地理解剩余的难点位于何处。程序员新手常常需要向经验丰富的程序员寻求帮助，但是如果他无法准确地描述自己所需要的帮助，那么对于双方都很可能是一种饱受挫折的体验。我们绝对不可能把问题削减为“这是我的程序，它无法工作。为什么？”使用问题削减技巧，我们可以准确地描述自己所需要的帮助，表示“这是我编写的一些代码。如您所见，我知道怎样计算两个三维坐标之间的距离，并且知道一个距离是否小于另一个距离。但是，我无法找到一种通用的解决方案，在众多坐标中寻找那对具有最小距离的坐标。”

1.2.6 寻找类比

对于我们而言，类比就是一个当前问题和一个已经解决的问题之间的相似性。我们可以利用这种相似性来解决当前问题，而这种相似性可能以多种形式存在。有时候，它意味着两个问题实际上是同一个问题。狐狸、鹅、玉米问题和 Quarrasi 锁问题就属于这种情况。

大多数类比并没有这么直接。有时候，相似性只涉及到问题的一部分。例如，两个数值处理问题可能在其他方面都不一样，唯一的共同点就是都需要比内置的浮点数据类型更精确的数值。我们无法使用这种类比来解决整个问题，但是如果我们已经想出了办法处理额外的精度问题，那就可以按照同样的方式再次处理相同的问题。

尽管认识类比是提高自己的问题解决速度和技能的最重要方式，但它也是最难培养的一种技巧。原因是在一开始很难发现类似的关系，除非有大量以前的解决方案可供参考。

这是成长中的程序员经常寻求捷径的地方。他们常常寻找那些与所需要的代码相似的代码，并在后者的基础上进行修改。但是，出于某些原因，这是错误的做法。首先，如果我们没有自己完成一个解决方案，就不能彻底理解并吸收它。简单地说，要想正确地修改一个自己并没有完全理解的程序是非常困难的。我们并不需要通过编写代码获得完全的理解，但是如果我们无法编写代码，我们对它的理解肯定是有限的。其次，我们所编写的每个成功的程序并不仅仅是一个当前问题的解决方案，它还是一种潜在的类比资源，可以供解决未来的问题所用。我们现在对其他程序员的代码的依赖程度越深，在未来仍然需要这种依赖的可能性也就越大。我们将在第 7 章深入讨论“良好的复用”和“不良的复用”。

1.2.7 试验

有时候，取得进展的最好方法是对事物进行试验并观察其结果。注意，试验与猜测并

不相同。当我们进行猜测时，自己输入一些代码并希望它能够完成任务，但对于能否达到目的自己并没有很强的信心。试验则是一种可控的过程。我们假设当某些代码执行时将会发生什么，然后对它进行试验，观察自己的假设是否正确。根据这些观察，我们可以获得一些信息，帮助自己解决原先的问题。

在处理应用程序编程接口或类库时，试验尤其能够提供帮助。假设我们编写了一个程序，使用了一个表示向量的库类（当元素被添加时能够自动增长的一维数组），但以前从来没有使用过这个类，并且不确定从这个向量中删除一个元素时会出现什么情况。在还没有掌握这个类的详细机制时，不要匆匆用它来解决原先的问题，而是可以先创建一个简短的单独程序，专门对这个向量类进行试验，尤其要在自己关心的场景下对它进行试验。如果在这个“向量演示程序”中花费一点时间，那么在以后的工作中它可以作为这个类的参考使用。

另一种形式的试验与调试相似。例如，一个特定的程序所产生的输出与预期的正好相反。如果输出的是数值并且所输出的数正如预想的一样，但顺序正好相反。如果在检查了代码之后还不明白为什么会发生这种情况，可以进行试验，修改代码，故意产生反向的输出（可能是运行一个反方向的循环）。如果输出结果发生了变化或者没有发生变化，都可能揭示出自己原先的源代码所存在的问题，为自己的思路打开一个缺口。不管怎样，我们都朝着解决方案迈进了一步。

1.2.8 避免陷入挫折感

最后一个技巧其实谈不上技巧，而是一句格言：避免陷入挫折感。当我们陷入挫折感时，自己的思维不再那么清晰、工作不再那么高效，所有的任务需要花费更长的时间并且看上去更为困难。更糟的是，挫折感会不断恶化，很可能从一开始轻度的焦虑变成最终难以遏制的烦躁。

当我向程序员新手提出这个忠告时，他们常常会反驳说，虽然他们在原则上同意我的观点，但他们对于是否会遭遇挫折是无法控制的。要求一位缺乏成功感的程序员避免陷入挫折感岂不是相当于要求小孩子在踩到钉子时不要叫喊吗？除非能够预料到自己将踩到钉子上，否则不可能及时抑制大脑的本能反应。因此，我们能够做的就是让小孩子避免踩到钉子。

程序员的处境并不相同。程序员不会像自我激励的大师一样大声叫喊，他们在遭受挫折时并不会对外部刺激做出强烈的反应。陷入挫折感的程序员并不是对显示器上的源代码

生气，尽管他可能会对着屏幕表达挫折感。反之，陷入挫折感的程序员是对自己生气。挫折的来源同时也是挫折的目标，也就是程序员的思维。

如果我们允许自己陷入挫折感时（我故意使用了“允许”这个词），实际上就为自己继续失败找到了借口。假设我们正在处理一个难题，并且挫折感不断上升。几个小时后，我们咬牙切齿，愤怒地把铅笔折成两截，并告诉自己如果能冷静下来，一定能取得实质性的进展。事实上，我们已经决定向自己的怒气屈服，觉得这要比勇敢面对这个难题容易得多。

最终，避免陷入挫折感是我们必须做出的决定。不过，我们可以采用一些思路，也许相助于实现这一点。首先，不要忘记第一条规则，也就是说始终要制订计划。虽然编写解决原先问题的代码是这个计划的目标，但这并不是这个计划的唯一步骤。因此，如果制订了一个计划并遵循了它，我们就能够取得进展并对此坚信不疑。如果完成了最初计划的所有步骤，但还是无法开始编写代码，这个时候就应该制订另一个计划。

另外，如果读者觉得继续干下去可能会陷入挫折感时，可以休息一会。一个诀窍是让手头上所处理的问题不止一个。这样，如果读者对一个问题感到无可奈何，可以把精力转向另一个问题。注意，如果成功地划分了问题，就可以对单个问题应用这种技巧，只要把陷入僵局的那部分问题扔在一边，转而解决其他部分的问题就可以了。如果没有可以处理的其他问题，也可以离开椅子做些其他事情。可以热热身，放松一下脑子，例如散步、洗衣服、做伸展运动（如果读者是一个整天坐在计算机前的程序员，我强烈建议养成做伸展运动的习惯）。在休息结束之前，不要再考虑那个问题。

1.3 习题

记住，为了真正学到东西，只有通过实践才有可能。因此，要尽可能多地完成习题。当然在第 1 章中，我们还没有讨论编程。但即使是这样，我还是鼓励读者尝试完成一些习题。读者可以把下面这些习题看成是演奏真正音乐之前的指法练习。

1.1 完成一个中等难度的数独题（可以从网络或当地报纸上寻找），用不同的策略进行试验，并对结果加以说明。能不能为解决数独问题编写一个通用计划呢？

1.2 考虑瓷砖滑块问题的一种变型，每块瓷砖上显示的是图片而不是数字。这种变化使难度增加了多少？为什么？

1.3 为瓷砖滑块问题寻找一种与我不同的解决策略。

1.4 搜索旧式的狐狸、鹅和玉米问题的变型并尝试解决它们。很多著名的难题来源于 Sam Loyd 或者是由他推广的。因此，可以通过他的名字进行搜索。而且，一旦发现（或者觉得太难而放弃思考，只是看看）了解决方案，思考怎样据此创建难题的简化版本。有哪些东西必须修改？仅仅修改约束条件还是描述方式？

1.5 尝试为其他传统的铅笔和纸游戏（例如纵横填字谜）编写一种显式的策略。应该从什么地方开始呢？在陷入僵局时应该怎么做呢？即使是“Jumble（类似七巧板的益智游戏）”这样的简单报纸游戏，也非常适合思考它的解决策略。

第 2 章

纯粹的难题

在本章中，我们开始讨论实际的代码。虽然后面几章要求读者具备中级编程水平，但本章所需要的编程技巧还是非常简单的。不过，这并不意味着本章所讨论的难题都非常简单，只不过我们能够把注意力集中在解决问题上而不是编程语法上。这是从最纯粹的角度探讨怎样解决问题。一旦明白了自己想要做什么，把自己的想法转换为 C++代码还是非常简单的。记住，光凭阅读本书能够获得的帮助还是非常有限的。我们应该对自己觉得有意义的问题进行深入的研究，在阅读我所提供的方法之前设法自己解决它们。在本章的最后，可以尝试完成一些习题，有许多习题是对本章所讨论的问题的延伸。

2.1 本章所使用的 C++简述

本章所使用的是基本的 C++，读者对此应该已经非常熟悉。它的内容包括控制语句 `if`、`for`、`while`、`do while` 和 `switch`。当然，读者可能还不熟悉怎样通过使用这些语句编写代码来解决原始问题，这正是本书将要探讨的。但是，读者首先应该理解这些语句的语

法，或者手头上准备一本好的C++参考书。

读者还应该知道怎样编写和调用函数。为了简单起见，我们将使用标准流cin和cout表示输入和输出。为了使用这两个流，需要在代码中包含必要的头文件iostream，并使用using指令引用这两个标准流对象：

```
#include <iostream>
using std::cin;
using std::cout;
```

为了简单起见，在代码清单中将不会出现这几条语句。我们假设所有使用了这两个流的程序都包含了这个头文件，并声明了这两条using指令。

2.2 输出图案

在本章中，我们将解决3个主要的问题。由于我们广泛地使用了问题的分治和削减技巧，因此这3个主要问题还将衍生出几个子问题。在第1节中，我们尝试编写一系列的程序，生成常规形状的图案输出。编写类似这样的程序需要使用循环这个技巧。

半个正方形

编写一个程序，只用两条输出语句cout << "#"和cout << "\n"，生成一个像半个5×5正方形形状（直角三角形）的#符号图案：

```
#####
####
###
##
#
```

这是说明约束条件重要性的另一个很好的例子。如果忽略只能使用两条输出语句（一条打印出一个#符号，另一条用于换行）这个要求，我们可以通过小林丸号的方法很简单地解决这个问题。但是，有了这个约束条件之后，我们必须使用循环来解决这个问题。

读者的脑海里可能已经有了这个问题的解决方案，不过我们假设读者暂时还不明白怎么解决这个问题。首先可供采用的一种优秀武器是削减法。我们怎样把这个问题削减为很容易

解决的情况呢？如果这个图案是完整的正方形而不是半正方形，情况会不会更简单一些？

一个正方形（半正方形问题的削减）

编写一个程序，只用 2 条输出语句 cout << "#"和 cout << "\n"，生成一个完整的 5 × 5 正方形形状的#符号图案：

```
#####
#####
#####
#####
#####
```

现在，这种情况足以让我们明白该如何做了。但是，假设我们仍然不知道怎么解决这样的问题，可以进一步削减这个问题，生成单行的#符号而不是正方形图案。

一行（半正方形问题的进一步削减）

编写一个程序，只用 2 条输出语句 cout << "#"和 cout << "\n"，生成一行由 5 个#符号所组成的形状：

```
#####
```

现在，我们所面临的就只是一个很简单的小问题，可以用一个 for 循环予以解决：

```
for (int hashNum = 1; hashNum <= 5; hashNum++) {
   cout << "#";
}
cout << "\n";
```

现在，我们可以返回到前一次削减，即完整的正方形。完整的正方形就是由 5 个#符号组成的单行图案的 5 次简单重复。我们知道怎样创建重复代码：只需要编写一个循环就可以了。因此，我们把单循环转换为双循环：

```
for (int row = 1; row <= 5; row++) {
   for (int hashNum = 1; hashNum <= 5; hashNum++) {
      cout << "#";
   }
   cout << "\n";
}
```

我们把前面的程序清单中的所有代码放入一个新的循环中，使它被重复执行 5 次，产生 5 个行，每行由 5 个#符号组成。现在，我们已经接近最终的解决方案了。我们应该怎样修改代码，使它产生半正方形的图案呢？如果我们观察上面这个程序清单并把它与自己所需要的半正方形的输出进行比较，可以发现问题在于条件表示式 `hashNum <= 5` 上。这个条件产生了 5 个相同的、由 5 个#符号所组成的行。我们需要一种机制，调整每行所生成的符号数量，使第 1 行为 5 个符号，第 2 行为 4 个符号，接下来依次类推。

为此，我们进行了另一个削减程序试验。把问题中比较麻烦的部分单独拿出来进行解决总是最为容易的。我们暂且把#符号放在一边，只讨论数字。

向下计数

在下面的程序清单中，在循环中的指定位置编写一行代码。这段代码显示数字从 5 到 1，每个数字出现在单独的一行中。

```
for (int row = 1; row <= 5; row++) {
   cout << ❶ expression << "\n";
}
```

我们必须找到一个 expression❶，在第 1 行时其值为 5，在第 2 行时其值为 4，接下来以此类推。如果我们需要一个当行号递增时其值递减的表达式，首先想到的可能是在行号前面加上负号，相当于把它乘以−1。这种方法可以产生递减的数字，却并不是我们所需要的数字。但是，我们可以进一步思考。我们所需要的值与行号乘以−1 所得之值的差是什么？表 2.1 对这个问题的分析进行了总结。

表 2.1 根据行变量计算所需要的值

行号	所需的值	行号*−1	行号与所需值之差
1	5	−5	6
2	4	−4	6
3	3	−3	6
4	2	−2	6
5	1	−1	6

差是一个固定值 6。这意味着我们所需要的表达式是 `row * -1 + 6`。稍微运用一下代数知识，可以把它简化为 `6 - row`。现在让我们试试：

```
for (int row = 1; row <= 5; row++) {
   cout << 6 - row << "\n";
}
```

很好，确实有效！即使这样还不行，所犯的错误也应该是微不足道的，因为我们采取了极为严谨的步骤。同样，用一段又短又简单的代码进行试验是极为容易的。让我们把这个表达式放在内层的循环中以作为它的限制条件：

```
for (int row = 1; row <= 5; row++) {
   for (int hashNum = 1; hashNum <= 6 - row; hashNum++) {
      cout << "#";
   }
   cout << "\n";
}
```

采用削减技巧，从问题描述到完整的程序之间需要更多的步骤，但每个步骤都变得非常容易。我们可以想像用滑轮组举起一件很重的东西。我们必须拉动更远的距离才能举起同样的重量，但每次拉绳子时所花费的力气却要小得多。

在继续解决前面这个问题之前，我们先来处理另一个形状问题。

侧三角形

编写一个程序，只用 2 条输出语句 cout << "#"和 cout << "\n"，产生一个类似侧三角形形状的#符号图案：

```
#
##
###
####
###
##
#
```

我们不打算再次讨论前一个问题经历的所有步骤，因为并无必要。“侧三角形”问题与“半正方形”问题类似，因此我们可以直接使用对前面那个问题进行研究所获得的成果。还记得“从自己所知道的开始”这句格言吗？我们首先列出从解决“半正方形”问题所掌握

的技巧，并把它们应用于这个问题。我们知道如下做法：

- 使用一个循环，显示一行特定长度的符号
- 使用嵌套循环显示一系列的行
- 使用代数表达式而不是固定值，为每一行创建不同数量的符号
- 通过试验和分析，发现正确的代数表达式

图2.1对我们当前所处的位置进行了总结。第1行显示了之前的“半正方形”问题。我们看到了所需要的#符号图案（a）、行图案（b）、正方形图案（c）以及把正方形图案转换为半正方形图案所需要的数字序列（d）。第2行显示了当前的“侧三角形”问题。我们同样看到了所需要的图案（e）、行图案（f）、矩形图案（g）和所需要的数字序列（h）。

```
#####                 #####   5
####                  #####   4
###         #####     #####   3
##                    #####   2
#                     #####   1

(a)         (b)       (c)     (d)

#                     ####    1
##                    ####    2
###                   ####    3
####        ####      ####    4
###                   ####    3
##                    ####    2
#                     ####    1

(e)         (f)       (g)     (h)
```

图2.1 解决形状问题所需要的各个组成部分

现在，我们生成（f）图案是没有任何问题的，因为它几乎与（b）相同。我们应该还能够生成（g）图案，因为它与（c）相似，只不过行数更多并且每一行的符号更少。最后，如果有人提供了能够产生数字序列（h）的代数表达式，我们就可以毫无困难地创建所需要的图案（e）。因此，我们知道为“侧三角形”问题创建解决方案所需的绝大部分脑力工作就只剩下：找出一个能够产生数字序列（h）的表达式。

因此，我们的注意力应该集中在这个地方。我们可以取“半正方形”问题的最终代码并进行试验，直到产生我们所需的数字序列。或者我们也可以进行猜测，创建一张如表2.1

的表格，看看自己的创造力如何。

现在让我们进行试验。在“半正方形”问题中，用一个较大的整数减去行号是可行的，因此我们看一下在一个行号从 1 到 7 的循环中用 8 减去行号会产生什么样的结果。图 2.2(b)显示了这个结果，但它并不是我们所需要的。我们应该怎么办呢？在前一个问题中，我们需要从大到小的数而不是从小到大的数，因此用一个较大的数减去循环变量就可以了。在这个问题中，我们先是从小到大然后再从大到小。从中间的数减去行号是否可行呢？如果我们把前面的 `8 - row` 替换成 `4 - row`，可以得到图 2.2（c）的结果。这个结果也不正确，但是如果忽略最后 3 个数左边的负号，它就是我们所需要的结果。如果我们使用绝对值函数去掉这些负号会怎么样？表达式 `abs(4 - row)` 产生结果如图 2.2（d）所示。现在，我们已经非常接近答案了，甚至已经闻到了它的味道。当我们需要先从小到大然后再从大到小排列的数列时，首先需要让它们从大到小然后再从小到大排列。但是，怎么才能通过这个数字序列提取出我们所需要的数字序列呢？

(a)	(b)	(c)	(d)	(e)
1	7	3	3	#
2	6	2	2	##
3	5	1	1	###
4	4	0	0	####
3	3	-1	1	###
2	2	-2	2	##
1	1	-3	3	#

图 2.2 解决“侧三角形”问题所需要的各个组成部分

让我们用一种不同的方式观察图 2.2（d）中的数字。如果我们是对空格而不是#号进行计数（如图 2.2（e）所示）会怎么样呢？（d）列就是我们对空格进行计数的正确的值序列。为了得到正确数量的#符号，可以把每行看成有 4 个格子，然后减去空格的数量。如果每行有 4 个格子，其中 `abs(4-row)` 为空格的数量，则具有#符号的格子的数量就可以用 `4-abs(4-row)` 获得。这种方法是可行的，可以把它插入到代码中，并进行试验。

我们已经通过类比避免了处理这个问题的大部分工作，并通过试验解决了剩余的问题。当一个新问题与我们已经解决的一个问题非常相似时，上面这套组合拳是极为有效的。

2.3 输入处理

前面的程序只产生输出。现在我们转换一下思路，尝试对输入进行处理。这类程序都

有一个共同的约束条件：输入将是逐字符读取的，程序必须在读取下一个字符之前处理完前一个字符。换言之，程序无法把字符存储在一个数据结构中并在以后进行处理，而必须在读取它们的同时对其进行处理。

在第一个问题中，我们将执行标识号验证。在现代生活中，几乎每样东西都有标识号，例如 ISBN 和客户编号。有时候这些标识号必须由手工输入，从而可能导致潜在的错误产生。如果一个错误输入的标识号与所有合法的标识号均不匹配，系统可以轻易地拒绝它。但是，如果这个标识号是错误的但同时又是合法的，那该怎么办呢？例如，出纳员为一位申请退货的顾客的账户进行记账时可能会错误地输入了另一位顾客的账号。为了避免这种情况的发生，在开发系统时我们要求能够检测出与标识号有关的错误。为了完成该任务，可以通过一个公式为一个标识号生成一个或多个额外的数字，使之成为扩展标识号的组成部分。如果这些数字发生了任何变化，标识号的原始部分和额外数字就不再匹配，因此这个标识号就会被拒绝。

Luhn 检验和验证

Luhn 公式是一种广泛使用的系统，用于对标识号进行验证。它根据原始标识号，把每隔一个数字的值扩大一倍。然后，把各个单独数字的值加在一起（如果扩大一倍后的值为 2 个数字，就把这两个数字分别相加）。如果相加之和可以被 10 所整除，那么这个标识号就是合法的。

编写一个程序，接受一个任意长度的标识号，并根据 Luhn 公式确定这个标识号是否合法。这个程序在读取下一个字符之前必须处理之前所读取的那个字符。

这个过程看上去有点复杂，但是只要通过一个例子就可以使之变得十分清晰。我们的程序只是对标识号进行验证，而不会创建检验数字。我们将讨论这个过程的起点和终点：计算一个检验数字以及对结果进行验证。图 2.3 演示了这个过程。在（a）部分，我们计算检验数字。原始标识号 176248 被圈在一个实线大方框内。从原始标识号的最右边开始（加上检验数字之后，它将成为从右边数过来的第 2 个数字）每隔一位的数字都扩大一倍。然后，把每个数字相加。注意，一个数字扩大一倍后如果变成了两位数，就分开考虑这 2 个数字。例如，当 7 扩大一倍成为 14 之后，就不是把检验和增加 14，而是分别加上 1 和 4。在当前例子中，检验和为 27，因此检验数字为 3，因为加上这个数字之后可以使总和为 30。记住，最终标识号的检验和应该能够被 10 所整除。换句话说，它应该以 0 结尾。

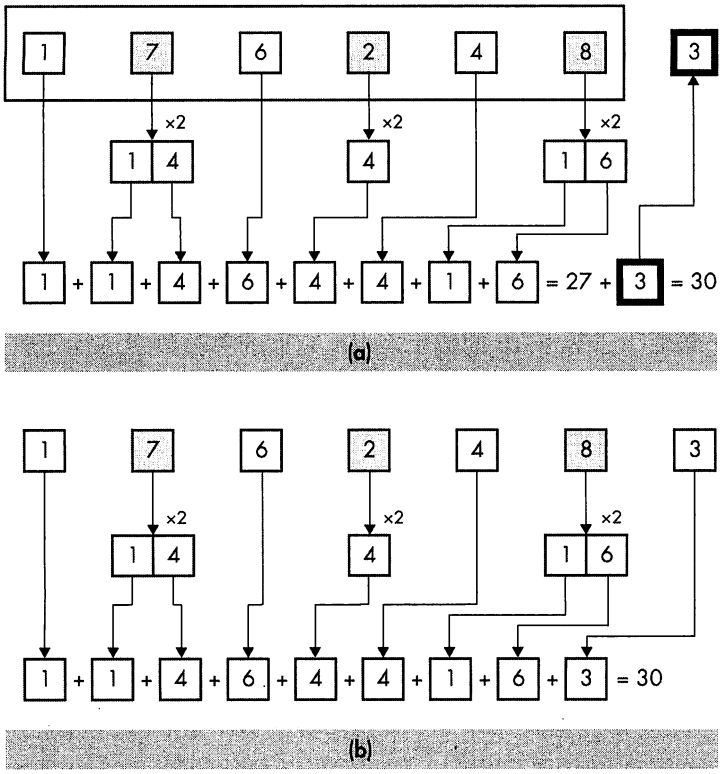

图 2.3 Luhn 检验和公式

在（b）部分，我们对 1762483 这个标识号进行验证，它现在已经包含了检验位。这是我们为解决上面这个问题所使用的步骤。和前面一样，我们从检验数字左边的那个数字开始把每隔一个数字的值扩大一倍，然后把包括检验数字在内的所有数字加在一起，以确定检验和。由于这个检验和可以被 10 所整除，因此这个标识号是合法的。

分解问题

解决这个问题的程序必须要处理几个独立的问题。其中一个问题是把数字扩大一倍。这个任务并不简单，因为需要扩大一倍的数字是从标识号的右端确定的。记住，我们并不是读取并存储所有的数字然后再对它们进行处理。我们需要在读取每个数字的时候就对它们进行处理。问题在于我们所得到的数字是从左到右排列的，但我们需要知道它们从右到左的排列，这样才能知道哪些数字需要扩大一倍。如果我们知道标识号有多少个数字，就知道哪些数字应该扩大一倍。但是，我们并不知道这个信息，因为问题描述中已经说明了标识号可以是任意长度的。另一个问题是扩大一倍后的数如果大于 10 就必须把这两个数字

分开处理。此外，我们还必须确定什么时候已经读取了整个标识号。最后，我们还必须知道怎样逐个数字地读取标识号。换句话说，用户将输入一个长长的标识号，但我们在读取的时候就当它是逐个数字输入的。

因为我们做事情总是需要一个计划，因此应该创建这些问题的一个列表，并逐个处理它们：

- 知道哪些数字需要扩大一倍
- 对扩大一倍后大于等于 10 的数字，根据它们的单独数字进行处理
- 知道已经到达了标识号的尾部
- 分别读取每个数字

为了解决这些问题，我们在编写最终的解决方案之前将对每个单独的片段进行处理。因此，我们并不需要按照任何特定的顺序处理这些问题。我们可以从最容易的问题开始，如果喜欢接受挑战，也可以从最困难的问题开始。当然，也可以从自己觉得最有趣的问题开始。

我们首先处理扩大一倍后大于或等于 10 的数。在这种情况下，问题的约束条件反而使事情变得简单而不是变得困难。计算一个任意整数的数字之和本身可能需要相当大的工作量。但是，相加后的值的范围是多少呢？如果我们从单独的数字 0～9 开始并把它们扩大一倍，最大值将是 18。因此，一共只有两种可能性。如果扩大一倍后的值为单个数字，就不需要再做处理。如果扩大一倍后的值为 10 或更大，它的范围肯定在 10～18 之间，因此第一个数字总是为 1。我们通过一个快速的代码试验来验证这种方法：

```
int digit;
cout << "Enter a single digit number, 0-9: ";
cin >> digit;
❶ int doubledDigit = digit * 2;
int sum;
❷ if (doubledDigit >= 10) sum = ❸1 + doubledDigit % 10;
else sum = doubledDigit;
❹ cout << "Sum of digits in doubled number: " << sum << "\n";
```

说明：%操作符被称为求模操作符。对于正数，它返回整数除法的余数。例如，12 % 10 的结果是 2，因此在 12 除以 10 之后，余数为 2。

这段代码非常简单：程序读取数字，把它的值扩大一倍❶，再对扩大一倍后的数的各个数字求和❷，然后输出求和的结果❹。这个试验的核心是对扩大一倍后大于 10 的数的各个数字的求和计算❸。和形状问题中一个特定的行所需要的#符号数量的计算一样，把这种计

算隔离到一个独立的小程序中可以使试验变得更简便。即使我们一开始无法得到正确的公式，也能够很快找到它。

在开始解决列表中的问题之前，我们首先把这段代码转化为一个短小的函数，这样就可以简化未来的代码清单：

```
int doubleDigitValue(int digit) {
   int doubledDigit = digit * 2;
   int sum;
   if (doubledDigit > 10) sum = 1 + doubledDigit % 10;
   else sum = doubledDigit;
   return sum;
}
```

现在，我们读取标识号的单独数字。当然，如果读者另有想法，也可以处理另一个不同的问题。但是，我觉得处理这个问题是很好的选择，因为它允许我们在测试问题的其他部分时能够很自然地输入标识号。

如果我们以数值类型（例如 int）的形式读取标识号，将会读取一个长长的数，需要处理很多事宜。另外，可以读取的最大整数也是有限制的。但在问题描述中，标识号可以是任意长度的。因此，我们必须逐字符读取。这意味着我们要知道怎样读取一个表示数字的字符并把它转换为整数类型，以便对它进行数学运算。如果我们把字符的值直接用于整数表达式中，会发生什么情况呢？观察下面的代码清单，它包含了示例输出。

```
char digit;
cout << "Enter a one-digit number: ";
❶ digit = cin.get();
int sum = digit;
cout << "Is the sum of digits " << sum << "? \n";

❷ Enter a one-digit number: 7
Is the sum of digits 55?
```

注意 我们使用了 get 方法❶，因为基本的提取操作符（例如在 cin >> digit 中）会跳过空白。这在当前并不会产生问题，但在以后可能会惹来麻烦。在示例的输入和输出中❷，我们就可以看到问题所在。所有的计算机数据在本质上都是数值类型的，因此单独的字符都是用整数字符码表示的。不同的操作系统可能使用不同的字符码系统，在本书中我们只关注常见的 ASCII 码系统。在这个系统中，字符 7 是以字符码值 55 存储的，因此当我们把这个字符作为整数时，得到的结果就是 55。我们需要一种机制把字符 7 转换为整数 7。

把字符数字转换为整数

编写一个程序，从用户那里读取一个表示0到9范围内数字的字符。把这个字符转换为0到9范围内对应的整数，然后输出这个整数以验证结果。

在前一节的形状问题中，有一个变量的值在某个范围之内，我们想把它转换为另一个范围内的值。我们创建了一张表，其中包含了原值和目标值，并检查了两者之差。这是一个类似的问题，我们可以再次使用这张表，如表2.2所示。

表2.2 字符码和目标整数值

字符	字符码	目标整数值	差
0	48	0	48
1	49	1	48
2	50	2	48
3	51	3	48
4	52	4	48
5	53	5	48
6	54	6	48
7	55	7	48
8	56	8	48
9	57	9	48

字符码和目标整数值之差始终是48，因此我们需要做的就是使字符码减去这个值。读者可能注意到它正是字符 0 的字符码。这个结果肯定是正确的，因为字符码系统总是从0开始按顺序存储数字字符的。因此，我们可以采用一种更通用、更易理解的解决方案，就是减去字符0的字符码而不是减去像48这样预先确定的值：

```
char digit;
cout << "Enter a one-digit number: ";
cin >> digit;
int sum = digit - '0';
cout << "Is the sum of digits " << sum << "? \n";
```

现在，我们转到问题的下一部分，判断哪些数字需要扩大一倍。这个问题可能需要几个步骤，因此我们首先试着对问题进行简化。如果一开始仅限于固定长度的数，应该怎么

做呢？这可以验证我们对基本公式的理解，同时又向最终的目标迈出一步。我们先试着把长度限制为 6，这个长度足以代表问题的总体难度。

Luhn 检验和验证，固定长度

编写一个程序，接受一个长度为 6 的标识号（包含检验数字），并根据 Luhn 公式检验和确定这个标识号是否合法。程序在读取下一个字符之前必须处理完当前字符。

和前面一样，我们甚至可以更进一步对问题进行简化，使之尽可能地简单。如果我们修改公式，不需要将任何数字扩大一倍，那该怎么做呢？此时程序只要读取各个数字并对它们进行求和就可以了。

简单的检验和验证，固定长度

编写一个程序，接受一个长度为 6 的标识号（包括检验数字），并根据一个简单的公式检验这个标识号是否合法。这个公式对每个数字进行求和，通过观察其和是否能够被 10 整除来检验它是否合法。程序在读取下一个字符之前必须处理完当前字符。

由于我们已经知道怎么以字符的形式读取单独的数字，因此可以相当轻松地解决这个固定长度的简单检验和问题。我们只需要读取 6 个数字，对它们进行求和，然后判断它们的和是否能被 10 所整除。

```
char digit;
int checksum = 0;
cout << "Enter a six-digit number: ";
for (int position = 1; position <= 6; position ++) {
   cin >> digit;
   checksum += digit - '0';
}
cout << "Checksum is " << checksum << ". \n";
if (checksum % 10 == 0) {
   cout << "Checksum is divisible by 10. Valid. \n";
} else {
   cout << "Checksum is not divisible by 10. Invalid. \n";
}
```

从现在开始，我们需要为实际的 Luhn 检验公式增加逻辑，也就是从右边数过来第 2 个

数字开始，把每隔一位的数字扩大一倍。由于我们当前仅限于处理 6 位长度的整数，因此需要把从左边开始位置分别为 1、3、5 的数字扩大一倍。换句话说，把位置为奇数的数字扩大一倍。我们可以使用求模操作符（%）确定奇数和偶数位置，因为偶数的定义是它能够被 2 所整除。因此如果表达式位置% 2 的结果是 1，这个位置就是奇数，应该把它扩大一倍。记住，在扩大一倍之后，如果结果大于或等于 10，还需要对这个结果的各个数字进行求和。这是前面的函数可以发挥作用的地方。当我们根据 Luhn 公式需要把一个数字扩大一倍时，只要把它传递给这个函数并使用它的结果就可以了。总之，我们只需要修改前面的代码清单中的 `for` 循环中的代码就可以了：

```
for (int position  = 1; position  <= 6; position++) {
   cin >> digit;
   if (position % 2 == 0) checksum += digit - '0';
      else checksum += doubleDigitValue(digit - '0');
}
```

到目前为止，我们在这个问题上已经取得很大的进展，但还需要完成一些步骤才能为任意长度的标识号编写代码。为了最终解决这个问题，我们需要采用分治法。假设需要修改前面的代码，使之适用于 10 位或 16 位的标识号。这个任务相当简单，只需要把表示循环上界的 6 修改为其他值就可以了。但是，假设我们要求对 7 位的标识号进行验证，这就需要一些额外的修改。因为如果标识号的数字个数为奇数并且我们是从右边第 2 位开始把每隔一位的数字扩大一倍，那么从左边开始的第 1 个数字就不需要扩大一倍。在这种情况下，我们需要把偶数位（2、4、6 等）的数字扩大一倍。我们暂时把这个问题放在一边，先考虑怎样处理长度为任意偶数的标识号。

我们所面临的第一个问题是怎样确定已经到达了标识号的末尾。如果用户输入了一个多位的标识号又按了 Enter 键表示结束，并且我们是逐字符读取输入的，那么在最后一个数字之后所读取的字符是什么呢？这实际上因操作系统而异，但我们只要试验一下就清楚了：

```
cout << "Enter a number: ";
char digit;
while (true) {
   digit = cin.get();
   cout << int(digit) << " ";
}
```

这个循环会一直运行，但它能够完成任务。如果输入 `1234` 并按下 Enter 键，其结果是 `49 50 51 52 10`（这个结果基于 ASCII 码，它可能因操作系统而异）。因此，10 就是

我们所寻找的结果。有了这个信息之后，我们就可以在前面的代码中用一个 while 循环代替 for 循环：

```
char digit;
int checksum = 0;
❶ int position = 1;
cout << "Enter a number with an even number of digits: ";
❷ digit = cin.get();
while ❸(digit != 10) {
  ❹if (position % 2 == 0) checksum += digit - '0';
   else checksum += 2 * (digit - '0');
  ❺digit = cin.get();
  ❻position++;
}
cout << "Checksum is " << checksum << ". \n";
if (checksum % 10 == 0) {
   cout << "Checksum is divisible by 10. Valid. \n";
} else {
   cout << "Checksum is not divisible by 10. Invalid. \n";
}
```

在这段代码中，position 不再是 for 循环的控制变量，因此必须对它进行初始化❶并单独增加它的值❻。现在，这个循环是由条件表达式❸所控制的，它用来检查字符码的值是否为行末符。由于我们需要一个值来检查这个循环的第 1 次迭代，因此在这个循环开始之前读取第 1 个值❷，并在循环执行了处理代码之后读取每个后续的值❺。

同样，这段代码将处理任意偶数长度的标识号。为了处理一个任意奇数长度的标识号，只需要对处理代码进行修改，反转 if 语句中的条件逻辑❹，把偶数位而不是奇数位的数字扩大一倍。

最后，要穷尽每种可能性。标识号的长度必须是奇数或偶数。如果我们预先知道长度，就可以知道应该把奇数位的数字还是偶数位的数字扩大一倍。但是，在读取完这个标识号之前，我们并不知道这个信息。有了这个约束条件，是不是无法解决呢？如果我们已经知道怎样解决奇数长度的问题和偶数长度的问题，但在完全读取之前并不确定标识号的位数，那么怎样才能解决这个问题呢？

读者可能已经看到了这个问题的答案。如果还没有看到，并不是因为答案很难，而是因为它隐藏在细节之中。我们在这里可以使用一种类比，但到目前为止还没有看到过相似的情况。我们可以创建自己的类比，设想一个明确描述这种特定情况的问题，看看直接面对这个问题时它是否能够帮助我们找到解决方案。把脑海里所形成的对这类问题的事先印象都清除掉，然后阅读下面这个问题。

正数或负数

编写一个程序，从用户那里读取10个整数。在输入了所有的整数之后，用户可能要求显示这些数中正数或负数的数量。

这是个简单的问题，看上去没有任何复杂之处。我们只需要一个对正数进行计数的变量，并用另一个变量对负数进行计数。当用户在程序的最后指定了具体的请求时，只需要显示适当的变量作为响应：

```
int number;
int positiveCount = 0;
int negativeCount = 0;
for (int i = 1; i <= 10; i++) {
   cin >> number;
   if (number > 0) positiveCount++;
   if (number < 0) negativeCount++;
}
char response;
cout << "Do you want the (p)ositive or (n)egative count? ";
cin >> response;
if (response == 'p')
   cout << "Positive count is " << positiveCount << "\n";
if (response == 'n')
    out << "Negative count is " << negativeCount << "\n";
```

它显示了我们在解决Luhn检验和问题时所需要用到的方法：同时以两种方式追踪当前的检验和，分别是在标识符为奇数长度和偶数长度的情况下。当我们读取完这个编号并确定了它的真正长度时，再选择表示正确检验和的变量。

片段汇总

现在，我们已经讨论了最初的“需要做”列表中的所有项目，可以把所有的内容都集中在一起来解决这个问题了。由于我们独立地解决了所有的子问题，因此已经准确地知道需要做些什么，并可以把前面的程序作为参考，很快生成最终的结果：

```
char digit;
int oddLengthChecksum = 0;
int evenLengthChecksum = 0;
int position = 1;
cout << "Enter a number: ";
digit = cin.get();
while (digit != 10) {
```

```
    if (position % 2 == 0) {
       oddLengthChecksum += doubleDigitValue(digit - '0');
       evenLengthChecksum += digit - '0';
    } else {
       oddLengthChecksum += digit - '0';
       evenLengthChecksum += doubleDigitValue(digit - '0');
    }
    digit = cin.get();
    position++;
}
int checksum;
❶if ((position - 1) % 2 == 0) checksum = evenLengthChecksum;
else checksum = oddLengthChecksum;
cout << "Checksum is " << checksum << ". \n";
if (checksum % 10 == 0) {
   cout << "Checksum is divisible by 10. Valid. \n";
} else {
   cout << "Checksum is not divisible by 10. Invalid. \n";
}
```

注意 当我们检查所输入的标识号的长度是奇数还是偶数时❶，把 position 减去 1，这是因为在循环中所读取的最后一个字符是表示结束的行末符，而不是这个标识号的最后一个数字。我们还可以把测试表达式写成(position % 2 == 1)，但它显然更容易产生混淆。换句话说，“如果 position - 1 为偶数，使用偶的检验和”这种表示方式要比“如果 position 是奇数，使用偶的检验和”更好，也更加符合逻辑。

这是到目前为止我们所看到的最长的代码清单，但我并不需要对代码中的任何内容进行注释或描述每个部分的工作原理，因为读者已经分别看过每个部分。这正是计划的威力所在。但是，值得注意的是，我所制订的计划并不是读者必须采用的计划。我在这个问题的最初描述中所看到的这些问题以及为了解决这些问题所采取的步骤，很可能与读者看到的问题和采取的解决步骤不同。读者的程序员背景以及工作经历决定了哪部分问题对于读者而言是小菜一碟、哪部分却是颇有难度，从而也会影响解决这个问题所需要采取的步骤。

在前一节中，读者可能觉得我在有些地方花了很大的篇幅解释其已经熟知的事实，但在有些对于读者而言颇感困惑的地方，我却轻描淡写地略过。此外，如果读者自己实现了这个问题的一个解决方案，可能会得到一个同等成功但看上去与我的程序截然不同的程序。对于问题而言，并不是只有一种“正确”的解决方案，满足所有约束条件的任何程序都可以看成是解决方案。并且对于任何解决方案，并不存在唯一正确的实现方法。

观察我们实现解决方案采取的所有步骤，并注意到最终代码的相对简洁之后，读者可能会忍不住在自己的问题解决过程中消减一些步骤。我必须对这种冲动提出一些告诫。采取多个步骤总比一次完成太多的任务要好得多，即使有些步骤看上去无足轻重。记住，我们的目标是解决问题。当然，首要目标是找到一个解决所陈述问题并满足所有约束条件的程序，其次是在尽可能少的时间内找到这个程序。尽量减少需要采取的步骤数量并不是我们的目标，没人会知道我们实际采取了多少个步骤。以登山为例，我们可以沿着坡度较小的蜿蜒山路到达山顶。如果忽略这条山路，而是从山脚直接往山顶爬，所需要攀登的直线距离当然短了很多，但它真的会更快吗？直接攀登的结果很可能在半途因为体力崩溃而放弃。

另外，记住解决问题的最后一个基本原则：避免陷入挫折感。每个步骤试图完成的工作越多，遭受挫折的可能性就越大。即使我们最后还是决定把一个困难的步骤分解为几个步骤，伤害可能已经产生，因为我们在心理上是觉得后退而不是取得进展。当我向程序员新手传授一步接一步的方法时，有时会听到学生的抱怨“嗨！这个步骤也太容易了点。”对此我的回复是“你在抱怨什么？”如果我们把一个一开始看上去非常困难的问题分解为几个很容易完成的片段，应该是件非常值得祝贺的事情，因为这正是我们希望看到的结果。

2.4 追踪状态

在本章中，我们所观察的最后一个问题也是最为困难的。这个问题具有大量不同的片段和复杂的描述，它充分说明了分解复杂问题的重要性。

消息的解码

一条消息被编码为一个文本流，被逐字符地读取。这个流包含了一系列由逗号分隔的整数，每个整数都可以用 C++的 `int` 类型表示。但是，一个特定整数所表示的字符取决于当前的解码模式。共有 3 种这样的模式：大写字母、小写字母和标点符号。

在大写字母模式下，每个整数表示一个大写字母：这个整数除以 27 的余数表示字母表中的具体字母（其中 1＝A，接下来以此类推）。因此，大写字母模式中的 143 这个值表示字母 H，因为 143 除以 27 的余数为 8，而 H 正是字母表中的第 8 个字母。

小写字母模式的机制类似，只不过表示的是小写字母。整数除以 27 的余数表示小写字母（1＝a，接下来以此类推）。因此，在小写字母模式下，56 这个值表示字母 b，因为 56 除以 27 的余数是 2，而 b 正是字母表中的第 2 个字母。

在标点符号模式下，是把整数除以 9 求余，表 2.3 给出了不同余数的解释。19 表示感叹号，因为 19 除以 9 的余数是 1。

在每条消息的开头，解码模式为大写字母。每当求余运算（除以 27 或 9，取决于具体的模式）返回 0 时，解码模式就会切换。如果当前模式为大写字母，就切换到小写字母模式。如果当前模式是小写字母，就切换到标点符号模式。如果当前模式是标点符号，就切换回大写字母模式。

表 2.3 标点符号解码模式

编号	符号
1	!
2	?
3	,
4	.
5	（空格）
6	;
7	"
8	'

和 Luhn 验证公式一样，我们将通过一个具体的例子清晰地说明所有的步骤。图 2.4 描述了一个解码的示例。顶部显示了最初的输入流。处理步骤是自顶向下进行的。（a）列显示了输入中的当前数字。（b）列是当前的模式，从大写字母（U）循环到小写字母（L），再到标点符号（P）。（c）列显示了当前模式的除数。（d）列是（a）列的当前输入除以（c）列的除数所得到的余数。（e）列显示了结果，它或者是个字符，或者在（d）列的结果为 0 时就切换到了循环中的下一个模式。

和之前的问题一样，我们首先可以明确地考虑创建解决方案所需要的技巧。我们需要读取一个字符串，直到读取到行末符。这些字符表示一系列的整数，因此需要读取这些数

字字符并把它们转换为整数以便进行处理。有了这些整数之后，需要把它们转换为单个字符进行输出。最后，我们需要一些方法处理解码模式，以便知道当前的整数应该被解码为小写字母、大写字母还是标点符号。我们首先把这些任务写成列表的形式：

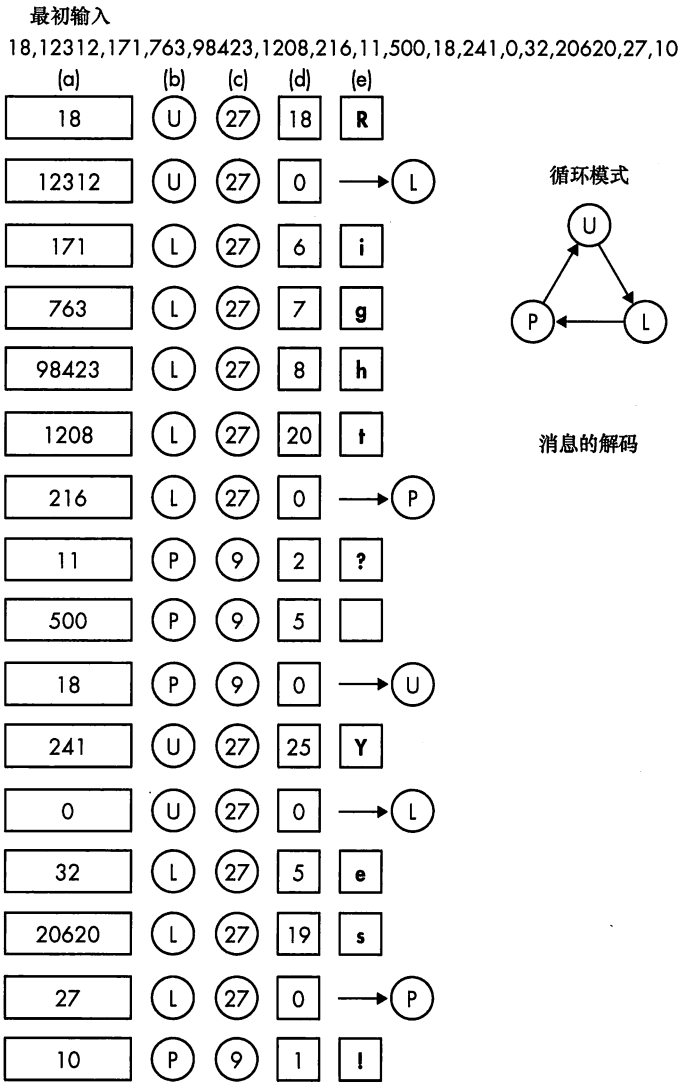

图 2.4 “消息的解码”问题的处理示例

- 逐个读取字符，直到读取了行末符。
- 把表示一个数的一系列字符转换为一个整数。

- 把一个 1～26 之间的整数转换为一个大写字母。
- 把一个 1～26 之间的整数转换为一个小写字母。
- 根据表 2.3 把一个 1～8 之间的整数转换为一个标点符号。
- 追踪一种解码模式。

在前面的问题中，我们大致已经了解应该怎样完成第一项任务。而且，尽管我们在 Luhn 验证公式中只处理单个的数字，但我认为在那个例子中所完成的任务对列表中的第二项任务也是很有帮助的。读者可能还记得 Luhn 算法的最终代码，不过就算因为时隔已久有所忘怀也没什么关系，只要回顾一下这些代码就可以了。一般而言，如果觉得当前问题的描述“似曾相识”时，可以发掘原有的类似代码，以帮助当前问题的研究。

我们接着讨论这些任务列表所剩余的任务。读者可能已经注意到，我把每个转换作为一个单独的任务。我觉得把一个数转换为小写字母与把一个数转换为大写字母是极为相似的，但转换为标识符号也许比较困难。不管是哪种情况，把问题分解得细一点并没有什么不好，只不过意味着我们从任务列表中所划去的内容会多一点而已。

保存代码供以后复用

现在的问题和前一个问题的元素之间的相似性显示了保存源代码供以后回顾的重要性。软件开发人员非常重视代码的复用，当我们利用旧软件的片段创建新软件时，就进行了代码的复用。这常常涉及到使用封装的组件或者原封不动地复用源代码。但是，重要的是能够方便地访问自己以前所编写的解决方案。即使我们并不是原封不动地复制旧代码，也可以复用以前所学到的技巧，而不需要重新学习它们。为了最大限度地利用这个优点，要设法保存自己编写的所有源代码（当然还要注意与客户或雇主所达成的知识产权协议）。

但是，能否从以前所编写程序中充分受益，很大程度上取决于这些程序是否被精心保存。如果找不到代码，显然就不能使用它。如果我们采用了一种一步接一步的方法，并编写单独的程序分别对不同的思路进行测试，最终才把它们集成为一个整体，就要确保保存所有的中间程序。当我们发现当前程序与曾经编写过测试程序的一个旧程序在某些领域存在相似之处时，就可以很方便地使用它。

让我们从整数到字符的转换开始。从 Luhn 公式程序中，我们知道需要读取一个 0～9 范围的字符数字，并把它转换为 0～9 范围的整数。我们怎么才能扩展这种方法，使之

能够处理多位数呢？让我们考虑最简单的可能性：两位数。这看上去非常简单。在两位数中，第一个数字是十位数，因此我们应该把这个数字乘以 10，然后与第二个数字所表示的值相加。例如，如果这个数是 35，以字符的形式分别读取了 3 和 5 之后，把它们分别转换为整数 3 和 5，然后通过表达式 3×10 + 5 得到总的整数。让我们用代码验证这一点：

```
cout << "Enter a two-digit number: ";
char digitChar1 = cin.get();
char digitChar2 = cin.get();
int digit1 = digitChar1 - '0';
int digit2 = digitChar2 - '0';
int overallNumber = digit1 * 10 + digit2;
cout << "That number as an integer: " << overallNumber << "\n";
```

这段代码可以达到目的。它输出了与我们的输入相同的两位数。但是，我们在扩展这个方法时还是遇到了一个问题。这个程序使用 2 个不同的变量保存 2 个字符输入，虽然它在当前不会有什么问题，但显然不适合作为一种通用的解决方案。如果采用这样的做法，我们所需要的变量数量就和输入的数字一样多。这很容易造成混乱，并且如果输入流发生了变化就很难修改输入数据的字数范围。对于把字符转换为整数这个子任务，我们需要一种更通用的解决方案。寻找这种通用解决方案的第一个步骤是对前面的代码进行限制，使它只能使用 2 个变量，1 个 `char` 变量和 1 个 `int` 变量：

```
cout << "Enter a two-digit number: ";
❶ char digitChar = cin.get();
❷ int overallNumber = (digitChar - '0') * 10;
❸ digitChar = cin.get();
❹ overallNumber += (digitChar - '0');
cout << "That number as an integer: " << overallNumber << "\n";
```

为了完成这个任务，我们在读取第 2 个数字之前首先完成对第 1 个数字的所有计算。在读取了第 1 个字符数字❶之后，我们把它转换为整数并乘以 10，然后存储结果❷。在读取了第 2 个数字❸之后，我们把它的整数值加到总和❹中。它的功能与前面的代码相同，区别在于只使用了 2 个变量，一个表示最近所读取的字符，一个表示整数的总值。下一个步骤是考虑怎样对这个方法进行扩展，使之适用于三位数。一旦完成了这个任务之后，我们很可能会发现一种模式，可以为任意位数的整数创建一个通用的解决方案。

但是，当我们尝试这种方法时还是遇到了一个问题。对于两位数而言，我们把左边的数字乘以 10，因为左边的数字总是十位数。三位数的最左边数字表示百分数，因此我

们需要把它乘以 100。接着，我们可以读取中间那个数字，把它乘以 10 并与总和相加。然后，我们再读取最后那个数字并把它与总和相加。这种方法当然可行，但无法指引我们找到通用的解决方案。明白问题所在吗？考虑前面的陈述：三位数最左边的那个数字是百位数。对于通用的解决方案，我们在读取逗号之前并不知道每个数有多少个数字。对于一个位数未知的数，它的最左边数字无法被确定是百位数或其他位数。因此，把每个数字加到总和之前，我们怎么确定它的乘数呢？我们是不是需要一种完全不同的方法呢？

像往常一样，当我们陷入僵局时，对问题的简化版本进行研究是一个非常好的思路。这里的问题是不知道将要处理的数有多少个数字。这类问题的最简单情况就是位数只有两种可能性。

读取一个三位数或四位数

编写一个程序，逐字符读取一个数，并把它转换为整数，只能使用 1 个 char 变量和 1 个 int 变量。这个数可能由 3 个或 4 个数字组成。

在读取结束之前不知道字符数量，却从一开始就需要进行计数有点类似于 Luhn 公式的问题。在 Luhn 公式中，我们并不知道标识号的长度为奇数还是偶数。在这种情况下，我们的解决方案是用两种不同的方式计算结果，并在最后选择正确的那种。在这里我们是不是可以采用类似的方法呢？如果这个数是 3 位或 4 位，就只有两种可能性。如果这个数有 3 个数字，最左边的那个数字就是百位数。如果这个数字具有 4 个数字，最左边的数字就是千位数。我们可以同时按三位数或四位数进行计算，并在最后选择其中正确的数。但是，在问题的描述中，我们只能使用 1 个数值变量。让我们放宽这个限制，以便于取得一些进展。

读取一个三位数或四位数，进一步简化

编写一个程序，逐字符读取一个数，并把它转换为整数，只能使用 1 个 char 变量和 2 个 int 变量。这个数可能由 3 个或 4 个数字组成。

现在我们可以把“双管齐下”的计算方法付诸实施。我们将用两种不同的方式处理前 3 位数字，然后看看是否存在第 4 位数字。

```
cout << "Enter a three-digit or four-digit number: ";
char digitChar = cin.get();
❶ int threeDigitNumber = (digitChar - '0') * 100;
❷ int fourDigitNumber = (digitChar - '0') * 1000;
digitChar = cin.get();
threeDigitNumber += (digitChar - '0') * 10;
fourDigitNumber += (digitChar - '0') * 100;
digitChar = cin.get();
threeDigitNumber += (digitChar - '0');
fourDigitNumber += (digitChar - '0') * 10;
digitChar = cin.get();
if ❸(digitChar == 10) {
    cout << "Numbered entered: " << threeDigitNumber << "\n";
} else {
  ❹fourDigitNumber += (digitChar - '0');
    cout << "Numbered entered: " << fourDigitNumber << "\n";
}
```

在读取了最左边的那个数字之后，我们把它的整数值乘以 100，并把它存储在表示三位数的变量中❶。我们还把这个数字乘以 1000，并把它存储在表示四位数的变量中❷。对于接下来的 2 个数字，也采用类似的模式进行处理。第 2 个数字在三位数中被当作十位数，在四位数中被当作百位数。第 3 个数字分别被当作个位数和十位数。在读取了第 4 个字符之后，我们把它与 10 这个数进行比较❸，确定它是否为行末符（和前一个问题一样，这个值可能因操作系统而异）。如果它是行末符，那么所输入的值就是个三位数。如果不是，我们需要把它作为最后一个数字添加到总和中❹。

现在，我们需要确定怎样去掉其中一个整型变量。假设我们完全去掉了 `fourDigitNumber` 变量，`threeDigitNumber` 仍然是被正确赋值的。但是，当我们需要 `fourDigitNumber` 时，就没办法再得到它了。使用 `threeDigitNumber` 的值，是否还有办法确定 `fourDigitNumber` 的值呢？假设原始输入为 1234。在读取了前 3 个数字之后，`threeDigitNumber` 变量的值将是 123，此时 `fourDigitNumber` 的值应该是 1230。一般而言，由于在计算过程中 `fourDigitNumber` 的每个乘数都是 `threeDigitNumber` 的对应乘数的 10 倍，因此前者的值总是后者的 10 倍。因此，我们只需要使用 1 个整型变量，因为在必要的情况下只要乘以 10 就可以得到另一个变量的值：

```
cout << "Enter a three-digit or four-digit number: ";
char digitChar = cin.get();
int number = (digitChar - '0') * 100;
digitChar = cin.get();
number += (digitChar - '0') * 10;
digitChar = cin.get();
number += (digitChar - '0');
```

```
digitChar = cin.get();
if (digitChar == 10) {
   cout << "Numbered entered: " << number << "\n";
} else {
   number = number * 10 + (digitChar - '0');
   cout << "Numbered entered: " << number << "\n";
}
```

现在我们已经有了一个可利用的模式。考虑把这段代码扩展到可以处理五位数。在计算了前 4 位数字的正确数值之后，我们可以重复与读取第 4 个字符相同的过程，而不是立即显示结果，也就是读取第 5 个字符，检查它是否表示行末符，如果是就显示之前计算所得的数，否则就把它乘以 10，再加上当前字符所表示的整数数值：

```
cout << "Enter a number with three, four, or five digits: ";
char digitChar = cin.get();
int number = (digitChar - '0') * 100;
digitChar = cin.get();
number += (digitChar - '0') * 10;
digitChar = cin.get();
number += (digitChar - '0');
digitChar = cin.get();
if (digitChar == 10) {
   cout << "Numbered entered: " << number << "\n";
} else {
   number = number * 10 + (digitChar - '0');
   digitChar = cin.get();
   if (digitChar == 10) {
      cout << "Numbered entered: " << number << "\n";
   } else {
      number = number * 10 + (digitChar - '0');
      cout << "Numbered entered: " << number << "\n";
   }
}
```

现在，我们可以轻松地对代码进行扩展，使它能够处理六位数或位数更少的数（即两位数）。这个模式是非常清晰的：如果下一个字符非行末符，就可以将当前的数乘以 10，然后与当前字符所表示的整数值相加。理解了这一点之后，我们就可以编写一个循环，处理任意长度的整数了：

```
cout << "Enter a number with as many digits as you like: ";
❶ char digitChar = cin.get();
❷ int number = (digitChar - '0');
❸ digitChar = cin.get();
  while ❹(digitChar != 10) {
     ❺number = number * 10 + (digitChar - '0');
     ❻digitChar = cin.get();
  }
  ❼cout << "Numbered entered: " << number << "\n";
```

在这里，我们读取第1个字符❶，并确定它的整数值❷。接着，我们读取第2个字符❸，并进入循环。我们在循环中检查最近所读取的那个字符是否为行末符❹。如果不是，就把当前为止的和乘以10，并与当前字符的整数值相加❺，然后再读取下一个字符❻。一旦读取了行末符，即表示当前整数的变量number就包含了可以输出的整数值❼。

这段代码用于处理一系列的字符到对应的整数值的转换。在最终的程序中，我们将读取一系列由逗号分隔的数。每个数必须单独读取并处理。和往常一样，一开始最好考虑一种能够说明这个问题的简单情况。让我们考虑 101，22[EOL]这个输入，其中[EOL]是为了清晰起见用来标识行末符的。对循环的测试条件进行修改，对行末符或逗号进行检查是很轻松的。接着，我们需要把处理一个数的循环放在一个更大的循环中，后者在所有的数被读取之前将一直持续。因此，内层循环在读取到[EOL]或逗号时将会结束，而外层循环只有在读取到[EOL]时才会结束：

```
❶ char digitChar;
  do {
      digitChar = cin.get();
      int number = (digitChar - '0');
      digitChar = cin.get();
      while ((digitChar != 10) && (digitChar != ',')) {
          number = number * 10 + (digitChar - '0');
          digitChar = cin.get();
      }
      cout << "Numbered entered: " << number << "\n";
  } while ❷(digitChar != 10);
```

这是说明多个小步骤重要性的另一个非常好的例子。尽管这是一个简短的程序，但是如果我们从头开始编写，双循环的“轮子套轮子”本质还是会让代码显得不够直观。但是，当我们从前一个程序出发再经过一个步骤实现这段代码时，它就变得简洁明了。digitChar的声明❶被移动至一行单独的代码中，使这个声明在整段代码中都位于作用域中。代码的剩余部分与前面的代码清单相同，区别只在于它被放在一个运行至读取了行末符才结束的do-while循环中❷。

有了这部分的解决方案之后，我们可以把注意力集中在处理单独的数上了。任务列表中的下一个任务是把一个范围在1～26之间的数转换为一个A～Z范围内的字母。稍微考虑一下，就可以发现它实际上是把单个数字字符转换为对应的整数值的逆操作。如果我们减去0的字符码，能够从0～9范围的字符码转换为0～9范围的整数值，应该也能够通过加上一个字符码，从1～26转换为A～Z。如果我们加上'A'会怎么样呢？下面是一次尝试，并显示了示例的输入和输出：

```
cout << "Enter a number 1-26: ";
int number;
cin >> number;
char outputCharacter;
outputCharacter = ❶number + 'A';
cout << "Equivalent symbol: " << outputCharacter << "\n";

Enter a number 1-26: 5
Equivalent letter: F
```

这显然是不正确的。字母表中的第 5 个字母是 E 而不是 F。出现问题的原因是我们从 1 开始的范围加上一个数的。当我们从另一个方向进行转换，把一个字符数字转换为对应的整数值时，我们所处理的范围应该是从 0 开始的。我们可以通过把 `number + 'A'`这个计算修改为 `number + 'A' - 1` 来修正这个问题。注意，我们可以查找字母 A 的字符码（在 ASCII 中为 65），并简单地使用这个值减去 1 的结果（例如，在 ASCII 码中为 `number + 64`）。但是，我更倾向于第一个版本，因为它更容易理解。换句话说，如果我们以后再查看这段代码，会觉得 `number + 'A' - 1` 要比 `number + 64` 容易理解得多，因为前者中'A'的出现可以提醒我们：它的意图是转换为大写字母。

在理清了思路之后，我们可以轻松地运用这个思路，只要把上面的计算❶修改为 `number + 'a' - 1`，就可以转换为小写字母。标点符号表的转换就没有那么简单，因为表中的标点符号在 ASCII 或其他任何字码符系统中都不是按照这个顺序出现的。因此，我们必须用穷举法处理这个问题：

```
cout << "Enter a number 1-8: ";
int number;
cin >> number;
char outputCharacter;
❶ switch (number) {
    case 1: outputCharacter = '!'; break;
    case 2: outputCharacter = '?'; break;
    case 3: outputCharacter = ','; break;
    case 4: outputCharacter = '.'; break;
    case 5: outputCharacter = ' '; break;
    case 6: outputCharacter = ';'; break;
    case 7: outputCharacter = '"'; break;
    case 8: outputCharacter = ❷'\''; break;
}
cout << "Equivalent symbol: " << outputCharacter << "\n";
```

在这里，我们使用了一条 switch 语句❶输出正确的标点符号。注意，这里使用了反斜杠作为转义符，以显示单引号❷。

把所有东西汇集在一起之前，我们还有一个子问题需要处理：当最近读取值的解码结果为0时，就进行模式的切换。记住，这个问题的描述要求我们取每个整数值除以27（如果当前是在大写模式或小写模式下）或9（如果当前是在标点符号模式下）的余数。当结果为0时，我们就切换到下一个模式。我们需要的就是一个存储当前模式的变量，并把逻辑放在“读取并处理下一个值”的循环中，在必要的时候切换模式。追踪当前模式的变量可以是个简单的整数，但是使用枚举显然可以使代码更容易理解。一个很好的经验是：如果一个变量只用于追踪一个状态，并且任何特定的值并没有内在的含义，那么使用枚举就是种很好的思路。在这个例子中，我们可以使用一个 int 型变量 mode，随意地设定1表示大写字母、2表示小写字母、3表示标点符号。但是，并没有实际的理由选择这些值。当我们以后回过头来阅读这段代码时，必须重新熟悉这个系统才能弄明白像 if (mode == 2)这样的语句表示什么意思。如果我们使用了枚举值，例如（mode == LOWERCASE）这样的语句，其含义就不言自明了。下面是根据这个思路所产生的代码，并包含了一个交互示例：

```
enum modeType {UPPERCASE, LOWERCASE, PUNCTUATION};
int number;
modeType mode = UPPERCASE;
cout << "Enter some numbers ending with -1: ";
do {
   cin >> number;
   cout << "Number read: " << number;
   switch (mode) {
      case UPPERCASE:
         number = number % 27;
         cout << ". Modulo 27: " << number << ". ";
         if (number == 0) {
            cout << "Switch to LOWERCASE";
            mode = LOWERCASE;
         }
         break;
      case LOWERCASE:
         number = number % 27;
         cout << ". Modulo 27: " << number << ". ";
         if (number == 0) {
            cout << "Switch to PUNCTUATION";
            mode = PUNCTUATION;
         }
         break;
      case PUNCTUATION:
         number = number % 9;
         cout << ". Modulo 9: " << number << ". ";
         if (number == 0) {
            cout << "Switch to UPPERCASE";
```

```
            mode = UPPERCASE;
          }
          break;
    }
    cout << "\n";
} while (number != -1);
```

```
Enter some numbers ending with -1: 2 1 0 52 53 54 55 6 7 8 9 10 -1
Number read: 2. Modulo 27: 2.
Number read: 1. Modulo 27: 1.
Number read: 0. Modulo 27: 0. Switch to LOWERCASE
Number read: 52. Modulo 27: 25.
Number read: 53. Modulo 27: 26.
Number read: 54. Modulo 27: 0. Switch to PUNCTUATION
Number read: 55. Modulo 9: 1.
Number read: 6. Modulo 9: 6.
Number read: 7. Modulo 9: 7.
Number read: 8. Modulo 9: 8.
Number read: 9. Modulo 9: 0. Switch to UPPERCASE
Number read: 10. Modulo 27: 10.
Number read: -1. Modulo 27: -1.
```

我们已经从任务清单上划掉了所有的任务，因此现在可以把这些单独的代码清单集成在一起，创建一个解决方案作为整体程序。我们可以通过不同的方法实现这种集成。我们可以只把两段代码放在一起，并在此基础上进行创建。例如，我们可以把读取并转换由逗号分隔的整数的代码与最近那段用于切换模式的代码组合在一起。接着，我们可以测试这个集成代码，并添加代码把每个数转换为对应的字母或标点符号。或者，我们可以从另一个方向创建解决方案，把从数字转换为字符的代码转换为一系列的函数，以便在 main 函数中调用它们。此时，我们在很大程度上已经把解决问题转移到软件工程领域，而后者是另一个不同的主题。我们创建了一系列的建筑块，这是最艰苦的工作，现在的任务是把它们装配在一起，如图 2.5 所示。

这个程序的每行代码几乎都是从出现在本节中的代码中抽取的。代码的主体❶来自于模式切换程序。中心处理循环❷来自于逐字符读取一系列逗号分隔的数字系列的代码。最后，我们可以看到把整数转换为大写字母、小写字母和标点符号的代码❹。少量的新代码由❸进行了标记。当最后一个输入是条模式切换命令时，`continue` 语句就跳到循环的下一次迭代，以跳过循环最后的那条 `cout << outputCharacter` 语句。

虽然这是一种剪切-粘贴工作，但它却是一种良好的剪切-粘贴工作，因为它复用了我们刚刚所编写的代码，因此我们已经完全理解了它。和以前一样，我们可以思考一下，相比直接编写最后的代码清单，分别完成每个步骤是多么的简单。毫无疑问，优秀的程序员即使不经过这些中间步骤也能产生最终的代码清单，但是这种做法会产生不必要的步骤、花

费不必要的时间，并且代码看上去会很丑陋，因为会有很多代码行在写完后被注释掉，之后又被添加回去。通过采取更小型化的步骤，所有的脏活都早早完成，代码绝对不会变得丑陋，因为我们每次所编写的代码绝对不会太长或太复杂。

```
❶  char outputCharacter;
   enum modeType {UPPERCASE, LOWERCASE, PUNCTUATION};
   modeType mode = UPPERCASE;
❷  char digitChar;
   do {
      digitChar = cin.get();
      int number = (digitChar - '0');
      digitChar = cin.get();
      while ((digitChar != 10) && (digitChar != ',')) {
         number = number * 10 + (digitChar - '0');
         digitChar = cin.get();
      }
      switch (mode) {
         case UPPERCASE:
            number = number % 27;
            outputCharacter = number + 'A' - 1;
            if (number == 0) {
               mode = LOWERCASE;
❸              continue;
            }
            break;
         case LOWERCASE:
            number = number % 27;
            outputCharacter = number + 'a' - 1;
            if (number == 0) {
               mode = PUNCTUATION;
               continue;
            }
            break;
         case PUNCTUATION:
            number = number % 9;
❹           switch (number) {
               case 1: outputCharacter = '!'; break;
               case 2: outputCharacter = '?'; break;
               case 3: outputCharacter = ','; break;
               case 4: outputCharacter = '.'; break;
               case 5: outputCharacter = ' '; break;
               case 6: outputCharacter = ';'; break;
               case 7: outputCharacter = '"'; break;
               case 8: outputCharacter = '\''; break;
            }
            if (number == 0) {
               mode = UPPERCASE;
               continue;
            }
            break;
      }
      cout << outputCharacter;
   } while (digitChar != 10);
   cout << "\n";
```

图 2.5 “消息解码”问题的汇总解决方案

2.5 结论

在本章中，我们观察了三个不同的问题。在某种意义上，我们采取了 3 条不同的路径来解决它们。在另一层意义上，我们每次采用了相同的路径，因为我们采用了相同的技巧把问题分解为几个部分，并编写代码分别解决每个部分的问题，然后利用从编写程序时所获取的知识或者直接使用程序中的代码行来解决原先的问题。在下一章中，我们将不会为每个问题都显式地使用这种方法，但是基本思路还是一样的：把问题分解为可管理的片段。

根据自身的背景，这些问题可能会在任何情况下出现，并且其难易程度也可能差异极大。

不管一个问题初看上去的难度如何，我总是推荐对面临的每个问题使用这种技巧。我们并不需要在遇到一个难度极大的问题时才尝试使用新技巧。记住，本书的其中一个目标就是增强自己解决问题的信心。经常通过“容易”的问题来练习这些技巧，这样在挑战困难的问题时才能够游刃有余。

2.6 习题

和以前一样，我建议尽可能多地尝试自身能够承受的习题。既然我们已经完全进入了实际的编程中，多完成一些习题对于提高自己解决问题的能力是非常重要的。

2.1 使用与本章前部分的形状程序相同的规则（只用两条输出语句，一条输出#号，另一条输出行末符），编写一个程序，生成下面的形状：

```
########
 ######
  ####
   ##
```

2.2 或者完成下面的形状：

```
   ##
  ####
```

```
  ######
 ########
 ########
  ######
   ####
    ##
```

2.3 下面是需要一些技巧才能完成的形状：

```
#              #
 ##          ##
  ###      ###
   ########
   ########
  ###      ###
 ##          ##
#              #
```

2.4 自行设计：考虑完成自己设计的#号对称图案，看看能否编写一个程序来生成它，并且遵循与上面相同的规则。

2.5 如果喜欢 Luhn 公式问题，尝试编写一个程序用于不同的检验数字系统，像 13 位的 ISBN 系统。这个程序可以接受一个标识符并对它进行验证，或者接受一个没有检验数字的数，并为它生成一个检验数字。

2.6 如果读者了解二进制数并知道怎样从十进制转换为二进制以及从二进制转换为十进制，尝试编写程序，完成长度不限的数的进制转换（但是可以假设这个数足以用标准 C++的 int 类型来表示）。

2.7 读者是否了解十六进制？尝试编写一个程序，让用户指定一个二进制、十进制或十六进制的输入，并用所有三种形式输出。

2.8 想要进行额外的挑战吗？对前一个习题的代码进行泛化，创建一个程序，把不超过十六进制的任何数转换为其他任何进制的数。例如，这个程序可以从九进制转换为四进制。

2.9 编写一个程序读取一行文本，对单词的数量进行计数并确定最长单词的长度、一个单词中最大数量的元音字母以及其他任何可以想到的统计数据。

第 3 章

用数组解决问题

在前一章中，我们所使用的变量仅局限于标量类型，也就是一次只能保存一个值的变量。在本章中，我们将讨论怎样使用最常见的聚合数据结构来解决编程问题，这种数据结构就是数组。尽管数组是一种存在基本局限性的简单类型，但它仍然极大地扩展了程序的功能。

在本章中，我们主要处理实际的数组，也就是通过基本的 C++语法所声明的数组，例如：

```
int tenIntegerArray[10];
```

但是，我们在这里所讨论的技巧也适用于具有类似属性的数据结构，其中最为常见的是 vector。vector 这个术语常常作为一维数组的同义词，但我们在这里用它表示一种更为具体的结构，它具有数组的属性，但没有指定最大元素数量。在我们的讨论中，数组具有固定的长度，而 vector 可以根据需要动态地增长或收缩。我们在本章中所讨论的每个问题都包含了一些约束条件，允许使用一种具有固定元素数量的结构。但是，对于没有这种约束

的问题，也可以使用 vector 来解决。

而且，在数组中所使用的技巧常常也可以用于那些并不具备上面所列出的每个属性的数据结构。例如，有些技巧并不要求随机访问，因此也可以用于像链表这样的结构。由于数组在编程中极为常见，并且数组技巧在非数组场合下也常常被使用，因此数组可以作为用数据结构解决问题的重要练兵场所。

3.1 数组基础知识概述

读者应该已经明白什么是数组，为了明确起见，我们在这里回顾一下数组的一些属性。数组是用一个名称所表示的相同类型的变量的集合，其中每个单独的变量都用一个编号来表示。我们把单独的变量称为数组的元素。在 C++和大多数其他语言中，第 1 个元素的编号为 0，但有些计算机语言的情况可能有所不同。

数组的主要属性直接来自于它的定义。一个数组所存储的每个值具有相同的类型，其他聚合类型可能存储混合类型的值。单独的元素是通过一个称为下标的编号所引用的。在其他数据结构中，单独的元素可能是由名称或键值所引用的。

根据这些主要属性，我们可以推断出几个次要的属性。由于每个元素是由一个从 0 开始的编号所指定的，因此我们可以轻松地访问数组中的每个值。在其他数据结构中，这可能是非常困难且低效的，甚至是不可能的。另外，在诸如链表这样的数据结构中，只能按顺序访问。数组提供了随机访问，意味着我们可以在任何时候访问数组中的任意元素。

这些主要和次要的属性决定了数组的使用方式。在处理任何聚合数据结构时，我们考虑问题的同时头脑中最好已经有了一组基本操作。我们可以把这些基本操作看成是常用工具，也就是数据结构的“锤子”、“螺丝刀”和“扳手”。并不是所有的机械问题都可以用这些基本工具来解决，但我们在遇到一个问题时总是先考虑是否可以用基本工具解决它，然后再考虑是否需要到五金商店购买其他工具。下面是我所认为的数组基本操作列表。

存储

这是最基本的操作。数组是一组变量的集合，我们可以对其中的每个变量进行赋值。为了把整数 5 赋值给前面所声明的数组的第 1 个元素，可以用以下的方式：

```
tenIntegerArray[0] = 5;
```

和其他变量一样，数组中元素的值在赋值之前是随机的“垃圾值”，因此数组在使用之

前应该初始化。在有些情况下（尤其是在测试时），我们想给数组中的每个元素赋一个特定的值。在声明数组时，我们可以用一个初始化值列表来完成这个任务。

```
int tenIntegerArray[10] = {4, 5, 9, 12, -4, 0, -57, 30987, -287, 1};
```

我们很快将见到初始化值列表的一种很常见的用法。有时候，我们并不想为每个元素赋不同的值，而是想把数组中的每个元素初始化为一个相同的值。这取决于具体的场合以及所使用的编译器，可以采用一些快捷方法把零赋值给数组中的每个元素（例如，除非另有指定，Microsoft Visual Studio 的 C++编译器会把任何数组中的每个元素都初始化为零）。但是，在当前这个阶段，在我们需要对数组进行初始化的场合，总是明确地对数组进行初始化，这可以提高代码的可读性。就像下面这段代码一样，它把一个数组的 10 个元素都初始化为−1：

```
int tenIntegerArray[10];
for (int i = 0; i < 10; i++) tenIntegerArray[i] = -1;
```

复制

我们可以创建数组的拷贝，它往往用于两种常见的情况。首先，我们可能想对数组进行大量的操作，但仍然需要保留数组原来的形式供以后处理。在经过操作之后再把数组恢复成原先的形式可能非常困难，如果更改了数组中的值，就更不可能。通过复制整个数组，就可以对数组的拷贝进行操作，而不需要破坏原先的数组。为了复制一个数组，只需要使用一个循环和一条赋值语句，就像初始化数值列表的代码一样：

```
int tenIntegerArray[10] = {4, 5, 9, 12, -4, 0, -57, 30987, -287, 1};
int secondArray[10];
for (int i = 0; i < 10; i++) secondArray[i] = tenIntegerArray[i];
```

这种操作也适用于绝大多数聚合数据结构。第二种情况则更倾向于数组所特有的。有时候，我们想把数据的一部分元素从一个数组复制到另一个数组，或者把一个数组的元素复制到另一个数组，作为一种重新排列元素顺序的方法。如果读者研究过归并排序，可能已经领略过这种思路。我们将在本章的后面内容中看到复制数组的例子。

提取和搜索

除了能够把值放入数组中之外，我们还需要能够把它们从数组中提取出来。从一个特定的位置提取一个值是非常简单的：

```
int num = tenIntegerArray[0];
```

搜索一个特定的值

但通常情况并没有这么简单。我们常常不知道所需要的位置，必须通过对数组进行搜索才能找到一个特定值的位置。如果数组中的元素并没有特定的顺序，最好执行线性搜索，即从数组的一端开始查看每个元素，直到找到所需要的值。下面是这种操作的一个基本版本：

```
❶ const int ARRAY_SIZE = 10;
❷ int intArray[ARRAY_SIZE] = {4, 5, 9, 12, -4, 0, -57, 30987, -287, 1};
❸ int targetValue = 12;
❹ int targetPos = 0;
  while ((intArray[targetPos] != targetValue) && (targetPos < ❺ARRAY_SIZE))
      targetPos++;
```

在这段代码中，有一个保存数组长度的常量❶、数组本身❷、保存了在数组中所找到的值的变量❸以及保存了所找到的值的位置的变量❹。在这个例子中，我们使用 ARRAY_SIZE 常量限制这个数组的迭代次数❺，这样即使在数组元素中并没有找到 targetValue 也不会越过数组的尾部。当然，我们也可以通过“硬编码”的方式用数字 10 代替这个常量，但是使用常量令代码更为通用，更容易被修改和复用。在本章的大多数例子中，我们将使用 ARRAY_SIZE 常量。注意，如果没有在 intArray 中找到 targetValue，targetPos 在循环结束之后将等于 ARRAY_SIZE。这足以说明情况，因为 ARRAY_SIZE 并不是合法的元素编号。但是，对这种情况进行检查的任务是由接下来的代码所完成的。另外，注意这段代码并没有采用任何方法处理目标值多次出现的可能性情况。当目标值第一次出现时，循环便告终止。

基于标准的搜索

有时候，我们所搜索的并不是一个固定的值，而是一个与数组中的其他值存在关系的值。例如，我们可能想要在数组中搜索最大值。我把完成这个任务的机制称为“山丘之王”（一种操场游戏），用一个变量表示数组到目前止所找到的最大值。用一个循环遍历数组中的所有元素，每当遇到一个比当前最大值更大的值时，就把以前的国王从山丘上踢下去并取而代之：

```
  const int ARRAY_SIZE = 10;
  int intArray[ARRAY_SIZE] = {4, 5, 9, 12, -4, 0, -57, 30987, -287, 1};
❶ int highestValue = intArray[0];
❷ for (int i = 1; i < ARRAY_SIZE; i++) {
      ❸if (intArray[i] ❹> highestValue) highestValue = intArray[i];
  }
```

highestValue 变量存储到目前为止在数组中所找到的最大值。在声明之时，它被赋值为数组中第一个元素的值❶，这样我们就可以从数组的第二个元素开始循环（可以从下标 1 而不是 0 开始）❷。在循环内部，我们把当前位置的值与 highestValue 进行比较，适时替换 highestValue 的值❸。注意，如果目标是找到最小值而不是最大值，只要把“大于”比较❹切换成“小于”比较就可以了（还需要更改变量的名称，以免产生混淆）。这种基本结构可以应用于所有需要观察数组中的每个元素，以找到最符合某个特定标准的值的场合。

排序

排序就是按特定的顺序排列数据。读者很可能已经遇到过一些数组的排序算法。这是性能分析的一个经典领域，因为有太多可供选择的排序算法，每种算法都存在因底层数据的特点而异的性能特征。对不同的排序算法的研究本身就可以作为整整一本书的主题，因此我们将不会过于深入地探讨这个问题。反之，我们把注意力集中在它的实际应用上。在大多数情况下，我们可以配备两种排序方法，一种是快速、易于使用的算法，另一种是优雅、易于理解的算法，当情况有变时可以充满信心地对其进行修改。对于快速、方便的算法，我们将使用标准库函数 qsort。遇到需要变通的场合，我们将使用一种插入排序算法。

用 qsort 进行快速方便的排序

C/C++程序员所采用的默认快速排序方法是标准库所提供的 qsort 函数（它的名称提示了底层的排序算法是一种快速排序，但这个库函数的实现并不一定要采用这种算法）。为了使用 qsort，必须编写一个比较函数。这个函数被 qsort 函数调用，用于比较数组中的两个元素，判断哪个应该出现在排序序列中的更前面。这个函数接受 2 个 void 指针。到目前为止我们还没有讨论过指针，但现在只需要知道应该把这两个 void 指针转换为指向数组元素类型的指针就可以了。这个函数应该返回一个整数，根据第一个元素是大于、小于或等于第二个元素，这个整数的值分别为正数、负数或零。具体返回的值无关紧要，重要的是它是大于、小于还是等于零。现在，我们通过采用 qsort 对一个包含 10 个整数的数组进行排序的简单例子来说明这种排序方法。以下是我们所使用的比较函数：

```
int compareFunc(❶const void * voidA, const void * voidB) {
  ❷int * intA = (int *)(voidA);
    int * intB = (int *)(voidB);
  ❸return *intA - *intB;
}
```

参数列表包含了 2 个 const void 指针❶。对于比较函数，一般都采用这种方式。这个函数内部的代码首先声明 2 个 int 指针❷，并把两个 void 指针转换为 int 指针类型。

我们也可以不采用这 2 个临时变量，在这里使用它们是为了清晰起见。关键在于，一旦完成了声明之后，intA 和 intB 将指向数组中的两个函数，*intA 和*intB 必须是两个可以进行比较的整数。最后，我们返回第一个整数减去第二个整数的结果❸。这样就产生了我们所需要的结果。例如，在 intA > intB 时我们想要返回一个正数，当 intA > intB 时，intA-intB 的结果是个正数。类似地，当 intB > intA 时 intA-intB 将是个负数，当两个整数相等时相减的结果为零。

有了比较函数之后，下面是 qsort 的一个示例用法：

```
const int ARRAY_SIZE = 10;
int intArray[ARRAY_SIZE] = {87, 28, 100, 78, 84, 98, 75, 70, 81, 68};
qsort(❶intArray, ❷ARRAY_SIZE, ❸sizeof(int), ❹compareFunc);
```

我们可以看到，对 qsort 的调用需要 4 个参数：需要排序的数组❶、数组的元素数量❷、数组元素的字节数（如此例所示，它一般是由 sizeof 操作符所确定的）❸，最后是比较函数❹。如果读者还不熟悉怎么把函数作为参数传递给其他函数，可以注意最后一个参数所使用的语法。我们所传递的是函数本身，而不是调用这个函数并传递调用的结果。因此，我们将简单地使用函数的名称，而不需要参数列表或括号。

容易修改的插入排序算法

在有些情况下，需要自己编写排序代码。有时候，内置的排序算法并不适用于自己所面临的情况。例如，我们需要根据另一个数组中的数据对一个数组中的数据进行排序。当我们必须自己编写排序算法时，可能需要一个简明并易熟练掌握的排序方法，可以根据需要对其进行修改。对于这类算法，合理的建议是使用一种插入排序。插入排序的工作方式与人们在打桥牌时所使用的理牌方式相似：一次抓起一张牌，把它插入到手里这把牌中的适当位置以维持整体的顺序，并移动其余的牌以留出空间。以下是整数数组的插入排序算法的基本实现：

```
❶ int start = 0;
❷ int end = ARRAY_SIZE - 1;
❸ for (int i = start + 1; i <= end; i++) {
      for (❹int j = i; ❺j > start && ❻intArray[j-1] > intArray[j]; j--) {
        ❼int temp = intArray[j-1];
          intArray[j-1] = intArray[j];
          intArray[j] = temp;
      }
  }
```

我们首先声明 2 个变量 start❶和 end❷，它们分别表示数组中第一个元素和最后

一个元素的下标。这种做法可以提高代码的可读性，如果需要还可以很方便地进行修改，只对数组的一部分元素进行排序。外层循环选择下一张需要插入的“牌”，它被插入到当前按升序排列的一把“牌”中❸。注意，这个循环把 i 初始化为 start + 1。记住，在“寻找最大值”的代码中，我们把最大值变量初始化为数组中的第 1 个元素，并从数组中的第 2 个元素开始循环。现在的代码所采用的思路是相同的。如果一共只有 1 个值（或“1 张牌”），按照定义就已经“排好了序”，然后开始考虑第 2 个值应该出现在第 1 个元素的前面还是后面。内层循环把当前值放在正确的位置，所采用的方法是不断地把当前值与它的前一个值进行交换，直到它到达正确的位置。循环计数器 j 是从 i 开始计数的❹，它在未到达数组的起点❺并且还没有找到这个新值的正确插入点之前对 j 进行递减❻。这个时候，我们使用 3 条赋值语句把当前值交换到数组中的前一个位置❼。换句话说，如果手上已经有 13 张牌，并且最左边的 4 张牌已经排好了序，可以把左起第 5 张牌放在正确的位置上，方法是不断将它移动一张牌的位置，直到除了左边之外不再有比它更小的值。这正是内层循环所执行的任务。外层循环为从最左边开始的每一张牌都执行这个任务。因此当外层循环完成之后，整个数组就已经排好了序。【译注：文中“最左边”是指数组的起始位置，“前一个位置”是指更靠近起点的那个位置】

在大多数情况下，插入排序并不是效率最高的排序方法。实事求是地说，即使对于执行插入排序而言前面的代码也不是效率最高的方法。它仅仅适用于长度较小或中等的数组。但是，它足够简单，很容易记忆，可以把它看成是一种定式思维。不管选择的是这种排序方法还是其他，我们应该有一种体面的或更好的排序方法，可以让自己在编写代码时充满信心。直接使用自己并不完全理解的、其他人的排序代码还是不够的。如果不明确具体的工作原理，就很难对它的行为进行调整。

计算统计数据

最后一种操作与提取操作相似，它也必须观察数组中的每个元素才能返回一个值。它与提取操作的区别在于它所返回的值并不是数组中的某个元素，而是根据数组中的所有值进行计算所产生的统计数据。例如，我们可以计算平均数、中位数或众数。在本章的后面内容中，我们将执行所有这些操作。一种常见的基本数据统计是计算一组学生成绩的平均值：

```
const int ARRAY_SIZE = 10;
int gradeArray[ARRAY_SIZE] = {87, 76, 100, 97, 64, 83, 88, 92, 74, 95};
double sum = 0;
for (int i = 0; i < ARRAY_SIZE; i++) {
    sum += gradeArray[i];
}
double average = sum / ARRAY_SIZE;
```

另一个简单的例子是数据验证。假设有一个称为 vendorPayments 的包含 double 值的数组，表示向销售商的支付情况。只有正值是有效的，因此如果出现了负值，就说明存在数据完整性问题。作为验证报告的一部分，我们可以编写一个循环，对数组中的负值进行计数：

```
const int ARRAY_SIZE = 10;
int countNegative = 0;
for (int i = 0; i < ARRAY_SIZE; i++) {
    if (vendorPayments[i] < 0) countNegative++;
}
```

3.2 用数组解决问题

一旦理解了数组的常用操作之后，就会发现解决数组问题与前一章所描述的解决简单数据的问题并没有太大的差别。我们首先讨论一个例子，并应用前一章所描述的技巧以及数组的常用操作。

问题：寻找众数

在统计学中，一组值的众数就是最常出现的值。编写代码，处理一个包含了调查数据的数组，确定这个数据集的众数。在这个数组中，接受调查者用 1～10 范围内的一个数表示一个问题的答案。对于我们而言，如果存在多个众数，可以任选其一。

在这个问题中，我们需要从一个数组中提取其中一个值。把搜索技巧作为类比，并从自己已经知道的知识出发，我们希望可以运用已经了解的提取技巧（提取数组的最大值）的一种变型。提取最大值的技巧就是把目前为止的最大值存储在一个变量里，然后把后续的每个值与这个变量进行比较，如果需要就替换这个变量。这里所采用的类比方法，就是把目前为止所看到的频率最高的值存储在一个变量里，每当在数组中发现一个出现频率更高的值时就替换这个变量的值。虽然听上去逻辑很清晰，但是反映到实际的代码时却会发现问题。我们首先观察这个问题的一个示例数组以及它的长度常量：

```
const int ARRAY_SIZE = 12;
int surveyData[ARRAY_SIZE] = {4, 7, 3, 8, 9, 7, 3, 9, 9, 3, 3, 10};
```

这些数据的众数是 3，因为 3 出现了 4 次，比其他任何值更为常见。但是，如果我们按照与处理最大值问题相同的方式按顺序处理这个数组，什么时候才能确定 3 是众数呢？当我们遇到 3 的第 4 次出现（也就是最后一次出现）时，怎么才能知道这确实是 3 在这个数组中的第 4

次也是最后一次出现呢？用单一的线性方式处理数组数据似乎没有办法解决这个问题。

因此，我们转向另一项技巧：简化问题。如果我们把同一个数所有的出现都聚集在一起，能不能简化问题呢？因此，假如示例数组的调查数据像下面这样排列：

```
int surveyData[ARRAY_SIZE] = {4, 7, 7, 9, 9, 9, 8, 3, 3, 3, 3, 10};
```

现在，2 个 7 出现在一起，3 个 9 也是如此，所有的 3 也都集中在一起。当数据按照这种方式排列之后，我们就能够线性地处理这个数组并找到众数了。用手工的方式处理这个数组，很容易对每个值的出现次数进行计数，因为我们只要保持计数，直到发现第一个不同的数。但是，把我们在大脑里的思路转换为编程语句时，可能需要一些技巧。因此，在编写这个简化后的问题的代码之前，我们先编写一些伪码。所谓伪码，就是与程序代码相似的语句，它们并不完全是日常语句，也不完全是 C++语句，而是折衷于两者之间。它可以提醒我们在编写每条语句时应该要完成什么任务。

```
int mostFrequent = ❶?;
int highestFrequency = ❶?;
int currentFrequency = 0;
❷ for (int i = 0; i < ARRAY_SIZE; i++) {
    ❸currentFrequency++;
    ❹if (surveyData[i] IS LAST OCCURRENCE OF A VALUE) {
        ❺if (currentFrequency > highestFrequency) {
            highestFrequency = currentFrequency;
            mostFrequent = surveyData[i];
        }
        ❻currentFrequency = 0;
    }
}
```

编写伪码并不存在“正确的”或“错误的”方法。如果决定使用伪码，就应该采用自己的风格。我在编写伪码的时候，对于完全有信心的地方，倾向于用传统的 C++语句来表示。对于那些还需要斟酌的地方，就用日常的语言进行描述。现在，我们知道需要用一个变量(mostFrequent)保存到目前为止所发现的频率最高的值。当循环结束的时候，如果我们的代码编写无误，它将保存这个数组的众数。我们还需要一个变量保存这个值的出现次数 (highestFrequency)，以便进行比较。最后，我们需要一个变量 (currentFrequency)，在按顺序处理数组时对当前所追踪的值的出现次数进行计数。上述这些变量在使用之前都应该进行初始化。对于 currentFrequency，在逻辑上它应该从 0 开始。但是对于其他变量，如果尚不知道其他代码，并不能确定应该初始化为什么值。因此，我们标上问号❶，提醒我们以后再做决定。

这个循环本身就是我们已经看到过的数组处理循环，因此这里列出了它的最终形式❷。

在循环内部，我们把对当前值的出现次数进行计数的变量的值加 1❸，然后到达基准点语句。我们需要检查，观察是否到达了一个特定值的最后一次出现❹。伪码允许我们暂时跳过具体的逻辑，审视剩下的代码。如果这已经是这个值的最后一次出现，我们就知道应该怎么做了，因为这就像“最大值”代码一样：观察这个值的计数是否比到目前为止的最大值更大。如果是，这个值就成为新的最常见值❺。然后，由于下一个被读取的值将是新值的第一次出现，因此我们把计数器重置为 0❻。

我们回到前面暂时跳过的 if 语句的逻辑。怎么才能知道一个值在数组中是最后一次出现呢？由于数组中的值已经分组，因此当数组中的下一个值与当前值不同时(用 C++的术语表示，就是当 surveyData[i]和 surveyData[i + 1]不相等时)，我们就知道现在是当前值的最后一次出现。并且，数组中的最后一个值也是某个值的最后一次出现，尽管它的后面不再有其他的值。我们可以通过判断 i == ARRAY_SIZE-1 这个条件来确定这个情况。如果这个条件为真，当前值就是数组中的最后一个值。

在理清了所有的脉络之后，我们就可以考虑变量的初始值问题。回顾“最大值”的数组处理代码，我们把“到目前为止的最大值”变量初始化为数组中的第一个值。现在，“当前的最常见”值用 2 个变量表示，mostFrequent 表示值本身，highestFrequency 表示它的出现次数。如果能把 mostFrequent 初始化为数组中所出现的第一个值并把 highestFrequency 初始化为这个值在数组中的出现次数当然是最好不过了，但在进入循环并开始计数之前，还没有办法确定第一个值的出现次数。此时，我们可能会想到，不管第一个值的出现次数是多少，它总是大于零。因此，我们把 highestFrequency 变量值初始化为零。当我们到达第一个值的最后一次出现时，这段代码就会把 highestFrequency 变量的值替换为第一个值的出现次数。完整的代码如下所示：

```
int mostFrequent;
int highestFrequency = 0;
int currentFrequency = 0;
for (int i = 0; i < ARRAY_SIZE; i++) {
    currentFrequency++;
  ❶// if (surveyData[i] IS LAST OCCURENCE OF A VALUE)
  ❷if (i == ARRAY_SIZE - 1 || surveyData[i] != surveyData[i + 1]) {
        if (currentFrequency > highestFrequency) {
            highestFrequency = currentFrequency;
            mostFrequent = surveyData[i];
        }
        currentFrequency = 0;
    }
}
```

在本书中，我们不会过多讨论纯粹的风格问题，例如文档（注释）的风格。但是，由于我们在这个问题中使用了伪码，因此在这里稍作说明。在伪码中保留为“日常语言”与最终代码中的注释具有密切的关系，并且日常语言本身也可以成为很好的注释。我已经在这段代码中证明了这一点。读者可能忘了这条 `if` 语句中的条件表达式❷的准确含义，但前一行的注释❶很清楚地对此进行了说明。

至于代码本身，它可以完成任务，但要求调查数据事先分组。对数据进行分组本身也算一项任务，不过如果我们对数组进行排序会怎么样呢？实际上我们并不需要对数据进行排序，但是排序可以实现我们所需要的分组。由于我们不想进行任何特殊类型的排序，因此只要在代码开始时简单地调用 `qsort` 就可以了：

```
qsort(surveyData, ARRAY_SIZE, sizeof(int), compareFunc);
```

注意 我们调用了与之前使用 `qsort` 时相同的 `compareFunc` 函数。有了这个排序步骤之后，我们就得到了最初问题的完整解决方案。因此，我们的任务就完成了，是不是这样？

重构

有些程序员谈到了会散发“不良气味”的代码。他们所讨论的代码并没有什么缺陷，只是在某些方面存在问题。有时候，这意味着代码过于复杂或者存在太多的特殊情况，使程序员难以对程序进行修改和维护。有时候，代码的效率无法达到预期，虽然对于测试用例而言没有问题，但是程序员还是担心把它应用于更大规模的数据时会出现性能问题。这也正是我所顾虑的地方。对于较小的测试用例，排序步骤几乎是瞬间完成的，但是如果数组极为庞大，又会出现什么情况呢？另外，我知道 qsort 函数可能使用的快速排序算法在数组中存在大量重复的值时会导致最差性能，但当前这个问题的关键是所有的值都位于 1～10 的范围之内，如果数组的规模很大，重复率可想而知。因此，我建议对代码进行重构。重构就是对代码进行改进，但并不影响它所执行的任务。我们需要一种即使对于巨大的数组仍然具有很高效率的解决方案，并且假设数组中所有的值的范围都在 1～10 之间。

再次思考我们所了解的数组操作。我们已经探索了一些版本的“寻找最大值”的代码。把“寻找最大值”的代码直接应用到 `surveyData` 数组并不会产生实用的结果。是不是存在这样一个数组，我们可以对它应用“寻找最大值”方法，从而得到调查数据的众数？答案是肯定的。我们所需要的数组是 `surveyData` 数组的柱状图。柱状图就是显示在底层数

据中的不同数据的出现频率的图形。我们所需要的数组就是表示这种柱状图的数据。换句话说，我们将在一个长度为 10 个元素的数组中存储 1 到 10 中的每个值在 surveyData 数组中的出现频率。以下是创建柱状图的代码：

```
const int MAX_RESPONSE = 10;
❶ int histogram[MAX_RESPONSE];
❷ for (int i = 0; i < MAX_RESPONSE; i++) {
      histogram[i] = 0;
  }
❸ for (int i = 0; i < ARRAY_SIZE; i++) {
     ❹histogram[surveyData[i] - 1]++;
  }
```

第一行代码声明了用于保存柱状图数据的数组❶。这个数组被声明为可容纳 10 个元素，但是调查答案的范围是在 1～10 之间，而数组的下标范围是 0～9。因此，我们必须对其进行调整，把 1 的计数放在 histogram[0]中，接下来以此类推。（有些程序员选择声明一个包含 11 个元素的数组，保留下标为 0 的位置，使每个计数都与它的逻辑位置对应。）我们采用一个循环，把数组的每个值初始化为零❷，然后就可以用另一个循环对 surveyData 数组中的每个值的出现次数进行计数了❸。我们必须仔细阅读这个循环中的语句❹。surveyData 数组中当前位置的值告诉我们应该增加 histogram 数组中哪个位置的值。我们通过一个例子更清楚地进行说明。假设 i 是 42。我们检查 surveyData[42]并发现这个值为 7（假设如此）。因此，我们需要把 7 的计数器的值加上 1。把 7 减去 1 得到 6，因此 7 的计数器位于 histogram 数组中下标 6 的位置，然后把 histogram[6]的值增加 1。

有了柱状图数据之后，就可以编写其余的代码了。注意，柱状图代码是独立编写的，这样就可以对它进行单独的测试。当问题可以很方便地分割为可以单独进行编写和测试的部分时，把它们放在一起反而不会节省什么时间。对上面的代码进行了测试之后，现在就可以搜索 histogram 数组的最大值了：

```
❶ int mostFrequent = 0;
  for (int i = 1; i < MAX_RESPONSE; i++) {
      if (histogram[i] > ❷histogram[mostFrequent]) ❸mostFrequent = i;
  }
❹ mostFrequent++;
```

尽管这个问题采用了“寻找最大值”代码的方法，但它还是存在不同之处。虽然我们搜索了 histogram 数组的最大值，但最终并不需要这个值本身，而是需要它的位置。换句话说，在我们的示例数组中，我们想知道 3 在调查数据中所出现的频率是否高于其他任何值，但 3 的实际出现次数并不重要。mostFrequent 将是 histogram 数组中最大值的位

置，而不是最大值本身。因此，我们把它初始化为 0❶而不是 location[0]的值。这还意味着在 if 语句中，我们把它与 histogram[mostFrequent]进行比较❷，而不是与 mostFrequent 本身进行比较。在找到一个更大的值时，我们把 mostFrequent 赋值为 1❸而不是 histogram[i]。最后，我们把 mostFrequent 的值增加 1，这正好与前一个循环中减去 1 得到正确的数组位置的操作相反。例如，如果 mostFrequent 告诉我们最大的数组位置是 5，表示调查数据中最常出现的项是 6。

柱状图解决方案的复杂度随着 SurveyData 数组的元素数量增加而线性增长，这也是我们能够期待的最好结果了。因此，相比原来的排序方法，它是更好的解决方案。但是，这并不意味着尝试前面那种方法是错误的或者是浪费时间的。当然，我们也可能直接就找到了现在这种方法，并没有经历前面那种方法，相当于走捷径直接到达了目的地。但是，如果读者觉得自己一开始所采用的解决方案并没有成为最终的解决方案，也不必因此而懊恼。编写一个最初的程序（记住，对于编写这个程序的人而言是最初的解决方案）其实是一种学习过程，我们不能期待前进的道路总是一帆风顺。另外，在一个问题上多走些弯路有助于我们在以后的问题中找到更便捷的道路。以当前这个特定的问题为例，注意，我们最初的解决方案（它的复杂度增长性对于这个特定的问题而言不够理想）对于调查答案并不严格局限于 1～10 这个狭小范围的问题而言就是正确的解决方案。如果我们想要找到一组整数值的中位数（中位数是位于中间的值，数据集中有一半的数大于这个值，另一半的数小于这个值），柱状图方法就没有办法做到这一点，但最早采用的那种寻找众数的方法却能够完成这个任务。

我们需要认识到，更曲折的道路并不是浪费时间，我们往往能够从中学到一些走最短路径时所无法学到的东西。这也是保存自己所编写的所有代码的原因之一，这样就可以很方便地找到它并在以后进行复用。即使是那种被证明是“死路一条”的代码，也可能会成为未来宝贵的资源。

3.3 固定数据的数组

在大多数数组问题中，数组对于程序而言是种外部数据来源，例如用户所输入的数据、位于本地磁盘上的数据或者来自服务器的数据。但是，为了最大限度地利用数组的功能，还需要了解其他可能使用数组的情况。有时候，需要创建一个在初始化之后元素的值不会再发生变化的数组。这种数组可以通过一个简单的循环甚至通过一种直接的数组查找方法来替换一整块控制语句。

在“对消息进行解码”问题的最终代码中，当处于“标点符号”模式时，我们使用了一条 switch 语句把解码后的输入数字（在 1～8 的范围之内）转换为适当的字符，因为数字与字符的联系是任意的。尽管这种方法可以完成任务，但它所产生的代码比大写字母和小写字母模式下对应的代码要长得多。而且，随着标点符号数量的增加，这种方法的伸缩性也不是很理想。我们可以使用数组代替 swith 语句来解决这个问题。首先，我们需要永久性地把标点符号保存在一个数组中，它们的出现顺序与编码方案中的顺序相同：

```
const char punctuation[8] = {'!', '?', ',', '.', ' ', ';', '"', '\''};
```

注意，这个数组被声明为 const，因此数组中的值是不会发生变化的。有了这个声明之后，我们就可以用一条引用数组内容的简单赋值语句替换整条 switch 语句：

```
outputCharacter = punctuation[number - 1];
```

由于输入的数字位于 1～8 的范围之内，但数组的下标是从 0 开始的，所以必须把输入数字减去 1 然后再将它作为下标引用数组。这和前面“寻找众数”程序的柱状图版本所采用的调整方式相同。我们也可以换个方向使用同一个数组。假设我们并不是对消息进行解码，而是对它进行编码。也就是说，我们所看到的是一系列需要被转换为数字的字符，可以采用原问题的规则进行编码。为了把一个标点符号转换为与它对应的数，必须在数组中找到这个符号的位置。这是使用线性搜索技巧所进行的提取操作。假设这个字符需要被转换并存储在 char 类型的变量 targetValue 中，我们可以按如下所示采用线性搜索代码：

```
const int ARRAY_SIZE = 8;
int targetPos = 0;
while (punctuation[targetPos] != targetValue && targetPos < ARRAY_SIZE)
    targetPos++;
int punctuationCode = targetPos + 1;
```

注意 就像在前一个例子中必须减去 1 才能得到正确的数组位置一样，我们在这个例子中必须把数组位置加上 1 才能得到标点符号的编号，把数组范围 0～7 转换为 1～8 范围内的标点符号码。尽管这段代码并不像单行代码这么简单，但它仍然要比一系列的 switch 语句要简单得多，并且其本身具有良好的伸缩性。如果我们把编码系统中的标点符号的数量扩大一倍，数组元素的数量也要扩大一倍，但代码的长度却并不需要增加。

一般而言，常量数组可以作为查找表，代替一系列笨拙的控制语句。假设我们编写了一个程序，计算某个州的营业执照价格。在这个州中，营业执照的价格因总销售额的数字

而有所不同。

表 3.1 营业执照的价格

商业分类	销售额门槛	执照价格
I	$0	$25
II	$50 000	$200
III	$150 000	$1 000
IV	$500 000	$5 000

对于这个问题，我们既可以用数组根据公司的总销售额确定它的商业分类，也可以用数组根据商业分类指定营业执照的价格。假设一个 double 类型的变量 grossSales 保存了一家公司的总销售额。根据销售数字，我们想把正确的值赋值给 int 类型的变量 category 和 double 类型的变量 cost：

```
const int NUM_CATEGORIES = 4;
❶ const double categoryThresholds[NUM_CATEGORIES ] =
      {0.0, 50000.0, 150000.0, 500000.0};
❷ const double licenseCost[NUM_CATEGORIES ] =
      {50.0, 200.0, 1000.0, 5000.0};
❸ category = 0;
❹ while (category < NUM_CATEGORIES &&
      categoryThresholds[category] <= grossSales) {
         category++;
      }
❺ cost = licenseCost[category - 1];
```

这段代码使用了 2 个包含固定值的数组。第一个数组存储了每种商业分类的总销售额门槛❶。例如，一家年销售额为$65 000 的公司的商业分类为 II，因为这个数额超过了分类 II 的门槛值$50 000，但是小于分类 III 的门槛值$150 000。第二个数组存储了每种分类的营业执照的价格❷。有了这个数组之后，我们把 category 初始化为 0❸并搜索 categoryThresholds 数组，当门槛值大于总销售额或者搜索到了分类数组的尾部时就停止搜索❹。不管是什么情况，当循环结束时，category 将根据总销售额被设置为 1～4 之间正确的值。最后一个步骤是使用 category 以引用 licenseCost 数组中的执照价格❺。和前面一样，我们必须对 1～4 范围的业务分类进行微调，与数组的 0～3 下标范围进行对应。

3.4 非标量数组

到目前为止，我们讨论了简单数据类型的数组，例如 int 和 double 类型。但是，程序员常

常需要处理复合数据类型的数组，例如结构和对象（struct 或 class）。尽管复合数据类型的使用多少会使代码变得复杂，但对于数组的处理，情况并不会变得更加复杂。数组处理通常只涉及结构或类的一个数据成员，我们可以忽略数据结构的其他部分。但是，有时候复合数据类型的使用可能要求我们对方法进行一些修改。

例如，思考寻找一组学生成绩的最大值问题。假设我们所看到的并不是一个 int 类型的数组，而是一个包含数据结构的数组，每个元素表示一条学生记录：

```
struct student {
    int grade;
    int studentID;
    string name;
};
```

处理数组时非常方便的一点是可以用字面值对整个数组进行初始化，使测试变得非常容易，即使是结构数组也不例外：

```
const int ARRAY_SIZE = 10;
student studentArray[ARRAY_SIZE] = {
    {87, 10001, "Fred"},
    {28, 10002, "Tom"},
    {100, 10003, "Alistair"},
    {78, 10004, "Sasha"},
    {84, 10005, "Erin"},
    {98, 10006, "Belinda"},
    {75, 10007, "Leslie"},
    {70, 10008, "Candy"},
    {81, 10009, "Aretha"},
    {68, 10010, "Veronica"}
};
```

这个声明表示 studentArray[0]用 87 表示它的 grade 值，用 1001 表示它的 studentID 值，用“Fred”表示它的 name 值。对于数组中的其他 9 个元素，也都进行了类似的初始化。至于代码的剩余部分就非常简单了，只要复制本章开始处的代码，用 studentArray[subscript].grade 替换 intArray[subscript]形式的每个引用。具体代码如下：

```
int highest = studentArray[0].grade;
for (int i = 1; i < ARRAY_SIZE; i++) {
    if (studentArray[i].grade > highest) highest = studentArray[i].grade;
}
```

由于现在每个学生包含了更多的信息，因此我们希望寻找成绩最好的学生的名字，而不是寻找成绩本身。这就需要对代码进行修改。当循环结束时，我们所得到的统计数据只是最佳成绩，无法据此直接确定这个成绩属于哪个学生。我们必须再次搜索数组，寻找具

有匹配成绩的元素，这似乎是多出来的额外工作。为了避免这个问题的出现，我们要么选择附加追踪与当前最大成绩值匹配的学生的名字，要么并不追踪最大成绩，而是选择追踪所找到的最大成绩在数组中的位置，这个方法与前面所采用的柱状图方法相似。后面这种方法是最通用的，因为追踪数组位置允许我们在以后提取该学生的任何数据。

```
❶ int highPosition = 0;
  for (int i = 1; i < ARRAY_SIZE; i++) {
      if (studentArray[i].grade > ❷studentArray[highPosition].grade) {
          ❸highPosition = i;
      }
  }
```

highPosition 变量❶的值是最大成绩的位置。由于并不是直接追踪最大成绩，因此当我们把到目前为止的最大成绩与当前成绩进行比较时，使用 highPosition 变量为下标以引用 studentArray 数组❷。如果当前数组位置的成绩更大，我们在处理循环中就把 highPosition 变量设置为当前位置❸。当循环结束时，我们就可以直接使用 studentArray[highPosition].name 访问具有最大成绩的学生的名字了，另外，还可以访问与这条学生记录有关的其他任何数据。

3.5 多维数组

到目前为止，我们只讨论了一维数组，因为它们是最为常见的。二维数组并不常见，三维乃至更多维的数组那就更为罕见了。这是因为大多数数据在本质上都是一维的。而且，多维的数据在本质上可以用多个一维数组来表示。因此，对多维数组的使用并不是程序员必须选择的方法。观察表 3.1 的营业执照数据，它就是一种很明显的多维数据，因为它是以网格形式展现的。但是，我们用了 2 个一维数组 categoryThresholds 和 licenseCost 表示这种多维数据。我们也可以用一个二维数组表示这张数据表，如下所示：

```
const double licenseData[2][numberCategories] = {
    {0.0, 50000.0, 150000.0, 500000.0},
    {50.0, 200.0, 1000.0, 5000.0}
};
```

很难看出把 2 个数组合并为 1 个数组有什么优点。代码并不会得到任何简化，因为没有理由同时处理所有的数据。反之，这种做法明显降低了代码的可读性以及数据的易用性。在原来的版本中，两个独立的数组的名称很清楚地说明了它们各自存储了什么数据。把这两个数组合并在一起之后，程序员必须用 licenseData[0][]这样的形式表示不同商业分类的销售门槛值，用 licenseData[1][]这样的形式表示营业执照的价格。

但是，有时候使用多维数组是合理的。假设我们需要处理3个销售代表的月销售数据，并且其中一个任务是找出任一代表的最高月销售额。把所有的数据存储在一个 3×12 的数组中意味着我们可以用嵌套循环一次性处理整个数组：

```
const int NUM_AGENTS = 3;
const int NUM_MONTHS = 12;
❶ int sales[NUM_AGENTS][NUM_MONTHS] = {
    {1856, 498, 30924, 87478, 328, 2653, 387, 3754, 387587, 2873, 276, 32},
    {5865, 5456, 3983, 6464, 9957, 4785, 3875, 3838, 4959, 1122, 7766, 2534},
    {23, 55, 67, 99, 265, 376, 232, 223, 4546, 564, 4544, 3434}
};
❷ int highestSales = sales[0][0];
for (❸int agent = 0; agent < NUM_AGENTS; agent++) {
    for (❹int month = 0; month < NUM_MONTHS; month++) {
        if (sales[agent][month] > highestSales)
            highestSales = sales[agent][month];
    }
}
```

尽管它直截了当地采用了基本的“寻找最大值”代码，但还是存在一些曲折之处。在声明二维数组的时候，注意初始化值列表是按照销售代表组织的，也就是一共分为 3 组，每组 12 个数据，而不是一共分为 12 组，每组 3 个数据❶。正如我们将在下一个问题中所看到的那样，这个决定可能会产生某种后果。和平常一样，我们把 highestSales 变量值初始化为数组中的第一个元素❷。读者可能会想到，在第一次遍历这个嵌套循环时，两个循环计数器都将是 0，这样我们就会把 highestSales 变量的这个初始值与它自身进行比较。这并不会影响结果，但有时候程序员新手会在内层循环体内添加第二条 if 语句，以避免这种轻微的低效情况：

```
if (agent != 0 || month != 0)
    if (sales[agent][month] > highestSales)
        highestSales = sales[agent][month];
```

但是，这种做法的效率要比前面的版本低得多。为了避免仅有的 1 次多余比较，我们付出了 50 次额外比较的代价。

另外，注意我使用了有含义的名称表示循环变量：agent 用于外层循环❸，month 用于内层循环❹。在处理一维数组的单循环中，使用这种描述性的标识符并没有多大的益处。但在处理二维数组的双循环中，有含义的标识符可以帮助我们清楚地区分维和下标，因为我可以查找并发现自己是在数组声明的 numAgents 那一维中使用了 agent。

即使使用了多维数组，有时候最好的方法还是一次只处理一维的数据。假设我们像前

面的代码一样使用了 sales 数组，要求显示销售额最高的那个销售代表的平均销售额。我们可以像前面一样用一个双循环来完成这个任务，但是如果自己把整个数组看成是 3 个独立的数组并单独对它们进行处理，代码就会变得更清晰并且更容易编写。

由于计算一个整数数组的平均数的代码可能会被反复使用，因此我们把它做成一个函数：

```
double arrayAverage(int intArray[], int ARRAY_SIZE) {
    double sum = 0;
    for (int i = 0; i < ARRAY_SIZE; i++) {
        sum += intArray[i];
    }
    double average = (sum + 0.5) / ARRAY_SIZE;
    return average;
}
```

有了这个函数之后，我们可以再次修改基本的“寻找最大值”代码，寻找具有最高平均销售额的代表：

```
int highestAverage = ❶arrayAverage(sales[0], 12);
for (int agent = 1; agent < NUM_AGENTS; agent++) {
    int agentAverage = ❷arrayAverage(sales[agent], 12);
    if (agentAverage > highestAverage)
        highestAverage = agentAverage;
}
cout << "Highest monthly average: " << highestAverage << "\n";
```

这里所采用的重要新思路是对 arrayAverage 函数的 2 次调用。这个函数所接受的第一个参数是个 int 类型的一维数组。在第一次调用时，我们向第一个参数传递 sales[0] ❶。在第二次调用时，我们向它传递了 sales[agent] ❷。因此，在这两种情况下，我们都只为二维数组 sales 的第一维指定了一个下标，但是并没有指定第二维。由于 C++中数组和地址之间的直接关系，这种引用方式表示指定行的第一个元素的地址，它可以被 arrayAverage 函数当作是一个只包含行的一维数组的基础地址使用。

如果上面的描述不太容易被理解，可以再次观察 sales 数组的声明，特别是它的初始化值列表。这些值在初始化值列表中的排列顺序与程序执行时它们在内存中的排列顺序相同。因此，值为 1856 的 sales[0][0]是首先出现的，然后是值为 498 的 sales[0][1]，接下来以此类推，直到第一个销售代表的最后一个月的数据 sales[0][11]，其值为 32。然后开始的是第二个销售代表的值，从值为 5865 的 sales[1][0]开始。因此，即使这个数组在概念上是共有 3 行并且每行包含 12 个值的二维数据，但它在内存中是以一个包含 36 个值的单一序列形式存在的。

值得说明的是，这种技巧之所以能够完成任务的原因在于我们在这个数组中放置数据的顺序。如果这个数组是按其他轴组织的，也就是说共包含 12 个月并且每月包含 3 个销售代表，就不能采用上面的做法。好消息是保证数组的组织形式是非常简单的，只要查看初始化值列表就可以了。如果我们想要单独处理的数据在数组的初始化值列表中并不是连续的，那么说明数组的组织形式有误。

关于这段代码，最后一点需要注意的是对临时变量 agentAverage 的使用。由于当前销售代表的平均月销售额有可能被引用两次；一次是在 if 语句的条件表达式中，另一次是在语句体的赋值语句中。因此，这个临时变量可以消除对同一个销售代表的数据调用 2 次 arrayAverage 函数的可能性。

把一个多维数组看成是一维数组的这种技巧符合把问题分解为更简单的组成部分这个原则。一般而言，这可以使多维数组问题在概念上容易被理解得多。尽管如此，读者仍然可能觉得这种技巧使用起来稍有难度。并且和大多数 C++程序员新手一样，读者可能对地址以及幕后的地址运算并不是非常得心应手。对于这样的感觉，我觉得最好的办法就是把概念的分离做得更加彻底，即把一个层次的数组放在一个结构或一个类中。假设我们创建了一个 agentStruct 结构：

```
struct agentStruct {
    int monthlySales[12];
};
```

既然已经不嫌麻烦创建了一个结构，我们还可以考虑添加其他数据，例如销售代表的标识号。但是从简化思路的角度而言，上面的代码就可以完成任务了。有了这个结构之后，我们就不需要创建一个二维数组 sales，而是创建一个包含销售代表的一维数组：

```
agentStruct agents[3];
```

现在，当我们调用求数组平均值的函数时，就不再需要使用 C++的特定技巧了，只要向它传递一个一维数组就可以了。例如：

```
int highestAverage = arrayAverage(agents[1].monthlySales, 12);
```

3.6 决定什么时候使用数组

数组只是一种工具。和其他任何工具一样，学习怎样使用数组的一个重要部分就是学习在什么时候应该使用它以及在什么时候不应该使用它。到目前为止，所讨论的示例问题

在它们的描述中已经假设了要使用数组。但是，在大多数情况下，我们并不能如此清晰地了解细节，必须自己确定是否需要使用数组。最为常见的情况是需要处理聚合数据但并没有告诉我们怎样组织数据。例如，在前面寻找众数的问题中，假设问题描述从“编写代码，处理一个包含调查数据的数组……”变成了“编写代码，处理一组调查数据……”现在，我们就必须自行决定是否使用数组。那么，我们应该怎样做决定呢？

记住，在创建数组之后，我们就无法更改它的长度。如果用完了空间，程序就会失败。因此，首先要考虑的是在编写处理聚合数据结构的程序时需要处理多少个值，或者至少对最大长度有比较准确的估计。这并不意味着我们必须在编写程序的时候知道数组的长度。和其他大多数计算机语言一样，C++允许我们创建在运行时确定长度的数组。假设众数问题进行了修改，我们事先并不知道总共有多少个调查答案，这个数字是在程序执行时由用户所输入的。我们可以动态声明一个数组来存储调查数据。

```
int ARRAY_SIZE;
cout << "Number of survey responses: ";
cin >> ARRAY_SIZE;
❶ int *surveyData = new int[ARRAY_SIZE];
for(int i = 0; i < ARRAY_SIZE; i++) {
    cout << "Survey response " << i + 1 << ": ";
  ❷cin >> surveyData[i];
}
```

我们使用指针记法来声明这个数组，通过调用 new 操作符对它进行初始化❶。由于 C++中指针和数组类型的互通性，即使 surveyData 被声明为指针，它的元素仍然可以使用数组记法进行访问❷。注意由于这个数组是动态分配的，在程序结束不再需要这个数组时，需要保证将其销毁：

```
delete[] surveyData;
```

和普通的 delete 操作符不同，delete[]操作符适用于数组。尽管对于整型数组，它并没有什么特别之处，但是如果创建了一个包含对象的数组，delete[]操作符可以保证数组中的单个对象在数组本身被删除之前被销毁。因此，对于动态分配的数组，应该养成使用 delete[]操作符的习惯。

不得不承担清理动态内存的责任是 C++程序员的难言之痛，但是如果选择了使用这种语言编程，那么就必须承担这个责任。程序员新手常常逃避这个责任，因为他们的程序通常很小，所执行的时间也很短，难以体会内存泄漏（内存不再被程序所使用，但是没有被销毁，将导致它们无法被系统的其他部分所使用）的恶劣后果。读者千万不要养成这种坏习惯。

注意 只有在预先得知调查答案的数量的前提下才能使用动态数组。考虑另一种情况：用户开始输入调查答案，但没有告诉我们答案的数量，而是通过输入-1（一种称为哨兵的数据项方法）表示已经输完了答案。我们是否仍然能够用数组来解决这个问题呢？

这属于灰色区域。如果我们能够保证调查答案的最大数量，仍然可以使用数组。在这种情况下，我们声明一个该长度的数组，暂时可以认为这样是安全的。但是，从长远来看，这种做法存在隐忧。如果调查人群的规模在未来扩大了怎么办？如果我们想对另一个调查人群使用同一个程序，会发生什么情况呢？按照更通用的说法，如果我们可以避免，为什么还要创建一个具有已知限制的程序呢？

更好的方法是使用没有固定长度的数据集合。如前所述，C++标准模板库中的 vector 类就可以作为一种能够根据需要增加长度的数组。一旦声明并初始化之后，vector 就可以和数组一样使用了。我们可以使用标准的数组记法为 vector 的一个元素赋值或者提取它的值。如果 vector 已经填满了它的最初长度并且需要再添加一个元素，可以使用 push_back 方法。用 vector 解决上面这个修改后的问题的代码如下：

```
❶ vector<int> surveyData;
❷ surveyData.reserve(30);
   int surveyResponse;
   cout << "Enter next survey response or -1 to end: ";
❸ cin >> surveyResponse;
   while (surveyResponse != -1) {
      ❹surveyData.push_back(surveyResponse);
       cout << "Enter next survey response or -1 to end: ";
       cin >> surveyResponse;
   }
❺ int vectorSize = surveyData.size();
   const int MAX_RESPONSE = 10;
   int histogram[MAX_RESPONSE];
   for (int i = 0; i < MAX_RESPONSE; i++) {
       histogram[i] = 0;
   }
   for (int i = 0; i < vectorSize; i++) {
       histogram[surveyData[i] - 1]++;
   }
   int mostFrequent = 0;
   for (int i = 1; i < MAX_RESPONSE; i++) {
       if (histogram[i] > histogram[mostFrequent]) mostFrequent = i;
   }
   mostFrequent++;
```

在这段代码中，我们首先声明了一个 vector❶并为调查答案保留了 30 个空间❷。第二个步骤从严格意义上讲并不是必需的，但是为 vector 保留稍多于可能保存的元素数量的空间，可以防止 vector 在我们向它添加元素时频繁改变长度。在进入数据项循环之前，读取第一个成绩❸，我们在前一章就已经使用过这个技巧，它允许我们在进行处理之前检查每个输入的值。在这个例子中，我们想避免把哨兵值−1 添加到这个 vector 中。调查结果是用 push_back 方法添加到 vector 中的❹。在数据项循环结束之后，我们使用 size 方法提取这个 vector 的长度❺。我们也可以在数据项循环中自己对元素的数量进行计数，但由于 vector 已经记录了自己的长度，因此可以避免这种重复劳动。这段代码的剩余部分与前面采用数组并且答案数量固定的版本相同，区别在于变量的名称有所不同。

但是，对 vector 的所有讨论都忽略了重要的一点。如果直接从用户处读取数据，而不是直接拥有一个包含数据的数组或其他数据结构，我们可能并不需要一个保存调查数据的数组，而是只需要一个保存柱状图的数组。我们可以在读取调查数据的时候处理它们。只有当我们在处理之前需要读取所有的数据或者需要多次处理数据时，才需要一个数据结构。这个例子显然并不属于这两种情况之一：

```
const int MAX_RESPONSE = 10;
int histogram[MAX_RESPONSE];
for (int i = 0; i < MAX_RESPONSE; i++) {
    histogram[i] = 0;
}
int surveyResponse;
cout << "Enter next survey response or -1 to end: ";
cin >> surveyResponse;
while (surveyResponse != -1) {
    histogram[surveyResponse - 1]++;
    cout << "Enter next survey response or -1 to end: ";
    cin >> surveyResponse;
}
int mostFrequent = 0;
for (int i = 1; i < MAX_RESPONSE; i++) {
    if (histogram[i] > histogram[mostFrequent]) mostFrequent = i;
}
mostFrequent++;
```

如果有以前的版本可以作为参考，上面的代码是很容易编写的。但是如果把用户数据读取到数组中并使用以前的处理循环语句，情况将会变得更加简单。这种“随时处理”方法的优点是它的效率。当我们只需要一次存储一个答案时，应该避免存储每个调查答案这种不必要的操作。基于 vector 的解决方案在空间上的效率是不高的：它所占据的空间要多于必要的空间，同时不会提供相应的好处。而且，除了处理所有的调查答案并在柱状图中

寻找最大值的循环之外，把所有的数据读取到vector中本身还需要一个循环。这意味着vector版本的工作量要大于前面的版本。因此，vector版本的时间效率也是不高的：它的工作量多于必要的工作量，但并没有提供相应的好处。在有些情况下，不同的解决方案具有不同的利弊，程序员必须在空间效率和时间效率之间做出决定。但是，在这个例子中，vector使程序在这两个方面效率都不高。

在本书中，我们不会花费太多的时间追踪每种低效率的情况。程序员有时候必须关注性能调校，这是对程序的时间和空间效率的系统性的分析和改进。对程序进行性能调校很像对赛车进行性能调校：这是一项讲究精确的工作，任何微小的调整都可能产生巨大的影响，需要深入理解幕后的工作机制。但是，即使我们没有时间、精力或专业知识对程序的性能进行完全的优化，还是应该避免那些可能降低整体效率的决定。不必要地使用 vector或数组并不相当于使用了一台燃油空气比过低的引擎，而是相当于在用一辆本田思域就可以装下所有行李的情况下却开一辆大巴去海滩度假。

如果我们确定需要多次处理数据，并且对数据集的最大长度相当有把握，那么决定是否使用数组的最后一个标准就是随机访问。以后我们将讨论像链表这样的数据结构，这种数据结构在根据需要进行增长方面与vector相似，但它只能按顺序访问，这点与数组和vector不同。也就是说，如果我们想要访问链表中的第10个元素，必须依次访问前9个元素才能访问它。相反，随机访问意味着可以在任何时间访问数组或 vector 中的任何元素。因此，最后一个规则是只有在需要随机访问的情况下才应该使用数组。如果我们只需要线性访问，可以考虑采用不同的数据结构。

读者可能注意到本章中的许多程序并不符合最后这个标准。虽然我们按顺序访问数组而不是随机访问数组，但我们仍然使用了数组。这就产生了对所有这些规则的常识性的期望。如果数组很小，前面这些负面影响都没什么关系。所谓的“小”可能因平台或应用程序而异。关键在于，那么程序所需要的一个集合中的数据少则为1多则为10，并且每个元素需要10个字节，那么分配一个长度为10的数组，有可能会浪费最多为90个字节的空间。但是，是否值得为节省这90个字节而寻找更好的解决方案呢？

要合理地使用数组，但所谓过犹不及，过分讲究完美反而是完美之敌。

3.7 习题

和平常一样，我建议读者尽可能地多完成一些习题。

3.1 是不是对正文中没有深入讨论排序有所失望？现在可以满足这个愿望。为了保证读者熟练使用 qsort 编写代码，请使用这个函数对一个包含了学生结构的数组进行排序。首先按成绩排序，接着按照 StudentID 进行排序。

3.2 重新编写寻找月平均销售额最高的代表的代码，寻找具有最高月销售额中位数的代表。如前所述，一组值的中位数就是“位于中间位置的值”，有一半的值大于这个值，还有一半的值小于这个值。如果值的数量为偶数，则中位数就是最中间两个值的简单平均。例如，在 10, 6, 2, 14, 7, 9 这组值中，最中间的两个值是 7 和 9。7 和 9 的平均值是 8，因此 8 就是这组值的中位数。

3.3 编写一个 bool 函数，它接受一个数组以及这个数组的元素数量为参数，判断这个数组中的数据是否已经排好了序。要求只对数组进行一次迭代就完成任务！

3.4 下面是常量值数组的一种变型。编写一个程序，创建一个置换密码问题。在置换密码问题中，所有的信息都是由大写字母和标点符号组成的。原来的信息称为普通文本，要求用其他字母置换每个字母来创建加密文本（例如，每个 C 被置换为 X）。对于这个问题，手工创建一个包含 26 个 char 元素的常量数组用于加密。让程序读取一段普通文本信息并输出对应的加密文本。

3.5 修改前面这个习题的程序，把加密文本转换回普通文本，验证加密和解码的正确性。

3.6 加大密码文本问题的难度，让程序随机地生成加密数组而不是使用手工编写的常量数组。这意味着把数组的每个元素设置为一个随机的值，但每个字母不能用于替换自身。因此第 1 个元素不能是 A，并且不能用同一个字母置换两个不同的字母。也就是说，如果第 1 个元素为 S，那么其他元素都不能是 S。

3.7 编写一个程序，向它提供一个整数数组，确定这个数组的众数，也就是数组中出现频率最高的那个数。

3.8 编写一个程序，处理一个学生对象数组，确定成绩的四分点，也就是进入前 25% 所需要的成绩、进入前 50%所需要的成绩和进入前 75%所需要的成绩。

3.9 考虑对 sales 数组进行修改：由于每年不断有新的销售人员加入以及旧的销售人员的离职，因此对于销售代表入职之前的月份以及离职之后的月份，销售额都被设置为-1。重新编写最高平均销售额或最高销售中位数的代码，抵消这些月份的影响。

第 4 章

用指针和动态内存解决问题

在本章中，我们将学习如何用指针和动态内存来解决问题。它们允许我们编写灵活的程序，处理数据长度在程序运行时才能确定的数据。指针和动态内存分配可以说是编程的“硬核”。当读者在编写程序时能够随时抓取内存块、把它们链接成实用的数据结构并在最后将它们清理干净不留残余时，就不再是编程的初学者，而是具备一定水准的程序员了。

由于指针比较复杂，并且许多流行语言（例如 Java）避开了对指针的使用，因此雏鸟初飞的程序员可能会说服自己完全跳过这个主题。这是错误的想法，指针和间接内存访问总是会在高级编程中使用，即使它们被隐藏在高级语言的机制幕后。因此，为了能够真正像程序员一样思考，必须学会用自己的方式思考指针和基于指针的问题。

但是，在开始解决指针问题之前，我们打算仔细梳理一遍指针的所有工作机制，不管是在表面还是在幕后。这样的学习可以提供两个益处。首先，这些知识可以使我们更有效率地使用指针。其次，揭开指针的神秘面纱之后，我们可以在使用它们的时候充满信心。

4.1 指针基础知识回顾

和前面的章节所讨论的主题一样，读者对基本的指针用法应该已经有所了解。但是，为了保证所有人都能跟上学习的步伐，我们对这些内容进行简单的回顾。

C++的指针是用星号（*）提示的。取决于具体的上下文环境，星号可能表示一个正在声明的指针，也可能表示被指向的内存而不是指针本身。为了声明一个指针，需要把星号放在类型名称和标识符之间：

```
int * intPointer;
```

它把变量 intPointer 声明为一个 int 类型的指针。注意星号是与标识符而不是与类型绑定在一起的。在下面的声明中，variable1 是个 int 类型的指针，但 variable2 是个普通的 int 类型的变量：

```
int * variable1, variable2;
```

变量前面的&符号可以作为取地址操作符。因此我们可以把 variable2 的地址赋值给 variable1：

```
variable1 = &variable2;
```

我们还可以把一个指针变量的值直接赋值给另一个指针变量：

```
intPointer = variable1;
```

最重要的是，我们可以在运行时分配只能通过指针访问的内存。这是用 new 操作符实现的：

```
double * doublePointer = new double;
```

从另一个方面通过指针访问内存的操作称为解引用，它是通过在指针左边加上星号实现的。它与指针声明中星号的出现位置相同。具体的上下文环境决定了不同的含义。下面是一个例子：

```
❶ *doublePointer = 35.4;
❷ double localDouble = *doublePointer;
```

我们把一个 double 值赋值给前面的代码所分配的内存❶，然后把这个值从这个内存位

置赋值给 localDouble 变量❷。为了销毁一块用 new 所分配的且不再需要使用的内存，可以使用 delete 操作符：

```
delete doublePointer;
```

这个过程的机制将在“内存细节”这一节中详细描述。

4.2 指针的优点

指针不仅允许我们进行静态的内存分配，还为我们提供了高效率地使用内存的新机会。使用指针的 3 个主要优点是：

- 运行时确定长度的数据结构
- 可改变长度的数据结构
- 内存共享

让我们详细讨论上述每个优点。

4.2.1 运行时确定长度的数据结构

通过使用指针，我们可以创建在运行时才能确定长度的数组，而不必在创建应用程序之前就确定数组的长度。这可以避免我们在“用尽数组的所有空间的潜在可能性”和“按照可能出现的最大长度创建数组，在平均情况下会浪费大量的空间”之间做出选择。前一章讨论“在使用数组时决定它的长度”时，我们第一次看到了运行时数据长度的概念。我们将在本章后面“可变长度的字符串”这一节中使用这个概念。

4.2.2 可改变长度的数据结构

我们还可以使基于指针的数据结构在运行时能够根据需要增长或收缩长度。最基本的可改变长度的数据结构是读者可能已经见过的链表。尽管这种结构中的数据只能按照线性顺序访问，但它所占据的空间就是其存储的数据所需要的，不存在浪费的空间。读者以后将会看到更为详细的、基于指针的其他数据结构具有顺序和“形状”，相比数组能够更好地反映底层数据之间的关系。因此，尽管数组所提供的完全随机访问是基于指针的数据结构

所无法提供的，但是基于指针的数据结构的提取操作（在结构中寻找最好地满足某个标准的元素）可能要快速得多。我们将在本章的后面内容中利用这个优点来创建一个可以根据需要增长的学生记录数据结构。

4.2.3 内存共享

指针可以通过内存块的共享提高程序的效率。例如，当我们调用一个函数时，可以通过引用参数向它传递一个指向一块内存的指针，而不是传递整块内存的一份拷贝。读者以前很可能看到过这种用法。在函数的形参列表中，引用参数的类型和名称之间有一个&符号：

```
void refParamFunction (int ❶& x) {
    ❷x = 10;
}

int number = 5;
refParamFunction(❸number);
cout << ❹number << "\n";
```

注意 &符号之前或之后的空格并不是必需的，这里加上空格的原因只是为了美观。在其他开发人员的代码中，可能会看到 int& x、int &x 甚至是 int&x 这样的写法。

在这段代码中，形式参数 x❶并不是实参 number❸的一份拷贝。反之，它是指向存储参数 number 的内存的引用。因此，当 x 发生变化时❷，参数 number 的内存空间也发生了变化，因此这段代码最后的输出是 10❹。引用参数可以作为向函数传递值的一种机制，如此例所示。更广泛地说，引用参数允许被调用函数和调用函数共享同一块内存，从而降低了开销。如果一个作为参数被传递的变量在内存中占据一千字节的空间，把这个变量作为引用进行传递意味着只要传递 32 或 64 字节而不是一千个字节。我们可以使用 const 关键字，表示使用引用参数的目的是为了追求性能而不是作为输出：

```
int anotherFunction(const int & x);
```

在引用参数 x 的声明中加上 const 前缀之后，anotherFunction 函数被调用的时候将接受调用者所传递的这个参数的引用，但不能修改这个参数的值，就像其他 const 参数一样。

一般而言，我们可以通过这种方式使用指针，允许程序的不同部分或程序中的不同数

据结构在不需要付出复制开销的情况下访问同一个数据。

4.3 什么时候使用指针

和数组一样，指针也存在一些潜在的缺陷，只能在适当的时候使用。我们怎么才能知道何时对指针的使用是适当的呢？在列出了指针的优点之后，我们可以说只有当自己需要利用指针的一个或多个优点的时候才应该使用它。例如，程序需要一个保存聚合数据的结构，但在程序实际运行之前无法准确地估计数据的数量时就可以使用指针。又如，当我们需要一个能够在程序执行时增长和收缩的数据结构时，也可以使用指针。或者，当程序中需要传递大型对象或其他数据块时，也可以使用指针来节省空间。但是，如果不是上述这几种情况，在使用指针和动态内存分配时就应该非常谨慎。

由于指针被认为是最难以使用的C++特性之一，如果并非必要，相信没有程序员会选择指针。但是，作者却发现过太多的反例。有时候，程序员会简单地陷入思维误区，以为指针是必须使用的。假设我们调用了一个由其他人所编写的函数，这个函数来自一个库或应用程序编程接口（API），也许具有下面这样的原型：

```
void compute(int input, int* output);
```

我们可能会觉得这个函数是用C而不是C++编写的，这也是它使用指针而不是引用(&)创造了一个“外向”参数的原因。在调用这个函数时，程序员可能会不小心使用了下面的代码：

```
int num1 = 10;
int* num2 = new int;
compute(num1, num2);
```

这段代码的空间效率不高，因为它创建了一个实际上并不需要的指针。它并不是使用2个整数的空间，而是使用了2个整数和1个指针的空间。这段代码的时间效率也不高，因为不必要的内存分配也需要消耗时间（下一节将对此进行说明）。最后，程序员必须记得删除被分配的内存。如果改用&操作符，上述这些缺点都可以得到避免。&操作符允许我们获取一个静态分配的变量的地址，如下所示：

```
int num1 = 10;
int num2;
compute(num1, &num2);
```

严格地讲，我们在后面这个版本的代码中仍然使用了指针，只不过是隐式地使用，不需要一个指针变量或者动态内存分配。

4.4 内存细节

为了理解动态内存分配是怎样实现运行时的大小改变和内存共享的，我们必须理解内存分配的基本工作原理。这是我认为有助于程序员新手学习 C++的领域之一。所有的程序员最终必须理解内存系统在现代的计算机中是怎样工作的，而 C++迫使我们必须直接面对这个问题。其他语言隐藏了内存系统的大量复杂细节，会使程序员新手觉得这些细节是无足轻重的，但这种想法是错误的。如果一切顺利，确实无需关心这些细节。但是，一旦出现了问题，对底层内存模型的缺乏理解就会在程序员和解决方案之间形成一条无法逾越的鸿沟。

4.4.1 堆栈和堆

C++在两个地方分配内存，分别是堆栈和堆。堆栈是有组织的并且非常整洁，而堆是离散的并且很杂乱。堆栈这个名称具有很强的描述性，它可以帮助我们领悟内存分配的连续性本质，思考如图 4.1（a）所示的木箱堆栈。存储一个木箱时，我们把它放在堆栈的顶部。为了从堆栈中移走一只特定的木箱，首先必须移除它上面所有的木箱。用现实的编程术语表示，意味着一旦在堆栈上分配了一块内存（也就是这个例子中的木箱），就没有办法改变它的大小，因为随时会有其他内存块堆在它的上面（它的上面叠放了其他木箱）。

在 C++中，可能需要在一个特定的算法中明确地创建自己的堆栈。但是，不管是什么情况，有一个堆栈是程序始终会使用的，那就是程序的运行时堆栈。每当一个函数被调用时（包括 main 函数），都会在运行时堆栈的顶部为它分配一块内存，这块内存称为活动记录。对活动记录的内容的完整讨论已经超出了本书的范围，但是作为问题解决者，需要理解活动记录就是变量的存储空间。所有的局部变量，包括函数的参数，都是在活动记录中分配的。让我们观察一个例子：

```
int functionB(int inputValue) {
  ❶return inputValue - 10;
}
int functionA(int num) {
```

```
    int localVariable = functionB(num * 10);
    return localVariable;
}
int main()
{
    int x = 12;
    int y = functionA(x);
    return 0;
}
```

在这段代码中，main 函数调用了 functionA 函数，后者又调用了 functionB 函数。图 4.1（b）显示了正好在执行 functionB 函数的 return 语句之前运行时堆栈的排列情况的简化版本❶。所有 3 个函数的活动记录将在一个连续内存的堆栈中排列，其中 main 函数位于堆栈的底部。（使事情变得更为复杂的是，堆栈在内存中从最高点开始朝较低位置向下生长也是可能的，而不一定是从较低位置生长到较高位置。但是，就算无视这种可能性，也不会产生任何危害。）从逻辑上说，main 函数的活动记录位于堆栈的底部，functionA 函数的活动记录位于它的顶部，而 functionB 函数的活动记录又位于 functionA 函数的顶部。在 functionB 函数的活动记录被销毁之前，另两个函数的活动记录是不可能被销毁的。

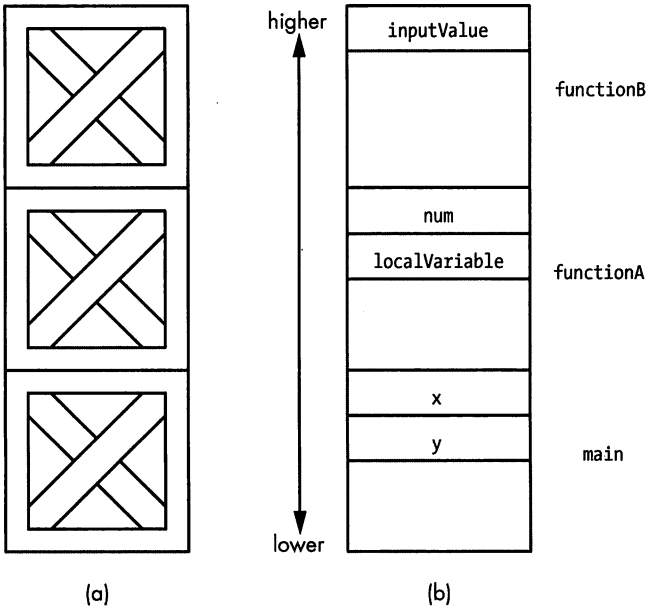

图 4.1 木箱堆栈（a）和函数调用堆栈（b）

堆栈具有高度的组织性，反之堆的组织性却很差。假设我们仍然需要堆放木箱，但木箱很脆，不能叠放在其他木箱上面。我们必须准备一个很大的、一开始为空的房间存放这些木箱，并且可以把它们放在地板上的任意位置。但是，木箱非常沉重，因此一旦放下之后，除非需要把它移出房间，否则就不想再移动它。与堆栈相比，堆这种存储系统优点和缺点并存。一方面，这种存储系统非常灵活，允许我们在任何时候获取任何木箱的内容。另一方面，房间很容易变得混乱不堪。如果所有的木箱都大小不一，要充分利用房间的空间就会变得特别困难。木箱之间可能会留下空隙，但又不足以放下另一只木箱。由于木箱很难移动，移走几只木箱常常会产生一些难以填充的空隙，而不是可以被充分利用的存储空间。用实际的编程术语表示，堆就像房间中的地板。一块内存是一系列连续的地址。因此，在执行许多内存分配和销毁的程序的生命期间，在剩余的已分配内存块之间会留下很多空隙。这个问题称为内存碎片。

每个程序具有自己的堆，可以在它上面分配内存。在C++中，这通常意味着调用new关键字，但是也可以调用旧式的像malloc这样的C函数进行内存分配。每次调用new（或malloc）会在堆上保留一块内存，并返回一个指向这块内存的指针。每次调用 delete（如果内存是用malloc分配的，则调用free）就会把这块内存返回到可用的堆内存池中。由于内存碎片的原因，内存池中并不是所有的内存都是同等有用的。如果程序首先在堆内存中分配了变量A、B和C，我们可以期望这些内存块是连续的。如果我们销毁了B，它所留下的空隙只有在以后所请求的内存小于或等于B时才能被分配，或者等到A或C被销毁之后。

如图4.2所示。在图4.2（a）部分，我们看到房间的地板上散布着木箱。在某个时刻，木箱可能摆放得比较整齐，但是随着时间的变化，木箱的排列就开始变得凌乱起来。现在，有一只小木箱图4.2（b）无法放到房间里的任何空隙中，尽管房间中未被利用的空间之和远远超过摆放这只木箱所需要的空间。在图4.2（c）部分，我们表示了一个小型的堆。虚线方框表示最小的内存块（不可分割），它可能是单个字节、一个内存字或者更大，这具体决取于堆管理器。阴影区域表示连续内存的分配。为了清晰起见，在进行内存分配时对它的内存块进行了编号。和充满细小空隙的地板一样，充满碎片的堆也有许多未分配的内存，这样就降低了它的利用率。未被使用的内存块总共有85个，但是如箭头所示，最大连续的、未使用内存只有17块。换言之，如果每个内存块的大小为1个字节，这个堆就无法满足任何大于17个字节的分配请求，即使它总共还有85个字节的未使用空间。

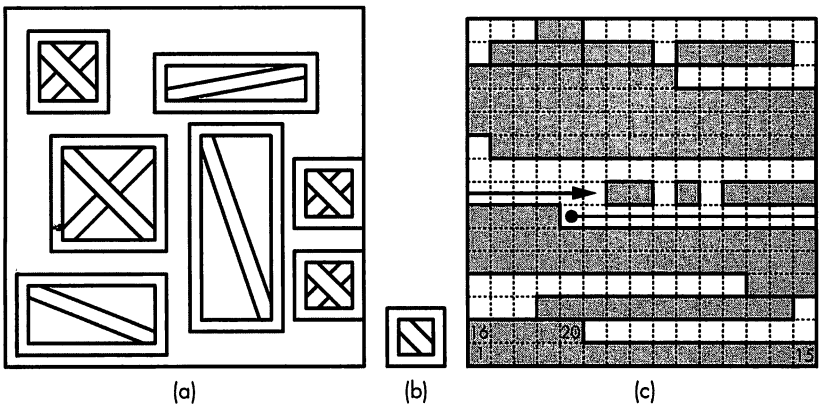

图 4.2 充满碎片的地板（a），不能再放下一只木箱（b）。（c）图是充满碎片的内存

4.4.2 内存的大小

第一个与内存相关的实际问题是如何限制它的使用，以确保只在必要时才分配内存。现代的计算机系统的内存数量相当庞大，很容易认为它是一个无限的资源。但是，每个程序实际上可以使用的内存数量还是有限的。另外，程序需要高效地使用内存，以避免系统的整体性能下降。在多任务操作系统（基本上涵盖了所有的现代操作系统）中，每个程序所浪费的每个字节的内存累加起来会导致当前所运行的程序没有足够的内存来运行。此时，操作系统就会把一个程序的内存块与其他程序的内存块进行交换，在短时间内会导致系统的性能大幅下降，这种情况称为系统颠簸（thrashing）。

注意，除了需要尽可能细地注意程序的总体内存使用情况之外，还要注意堆栈和堆都具有空间上限。为了证明这一点，我们可以在堆上一次分配一千个字节，直到无法继续：

```
const int intsPerKilobyte = 1024 / sizeof(int);
while (true) {
    int *oneKilobyteArray = new int[intsPerKilobyte];
}
```

必须强调一下，编写这段可怕代码只是为了证明一个事实。如果想要在自己的系统中试验这样的代码，首先应该保存所有的工作，以确保安全。应该出现的结果是程序终止，并且操作系统表示这段代码已经生成但没有处理一个 bad_alloc 异常。这个异常是在调用 new 进行内存分配但堆中所有未分配的内存块都无法满足这次分配所请求的内存数量时被抛出的。用尽堆内存的现象称为堆溢出。在有些系统中，堆溢出可能很常见，但在有些系统中，在程序产生 bad_alloc 异常之前早已出现了系统颠簸（在我所使用的系统中，进行前

面的调用时，在分配2G字节之前不会导致new调用的失败）。

运行时堆栈也会出现类似的情况。每个函数调用在堆栈上分配空间，并且每个活动记录都需要一定数量的固定开销，即使是那些不接受任何参数并且没有使用任何局部变量的函数。证明这一点的最简单方法是用一个失控的递归函数：

```
❶ int count = 0;
  void stackOverflow() {
     ❷count++;
     ❸stackOverflow();
  }
  int main()
  {
    ❹stackOverflow();
     return 0;
  }
```

这段代码中有一个全局变量❶，在大多数情况下这是一种不良风格，但我们在这里需要一个值可以贯穿使用于所有的递归调用。当这个变量在函数的外部声明时，系统不会在函数的活动记录中为它分配内存，另外也不存在其他任何局部变量或参数。这个函数所执行的任务就是把count的值增加1并进行一次递归调用❸。递归将在第6章深入讨论，这里简单地提到递归，因为它可以尽可能长地创建调用链。函数的活动记录位于堆栈上，直到此函数结束。因此，在main函数中第一次调用stackOverflow函数时，会在运行时堆栈上创建一条活动记录，它在第一个调用结束时才会被删除。但是，这种情况永远不会发生，因为这个函数再次调用了stackOverflow函数，并在堆栈上创建了另一条活动记录，然后第三次调用stackOverflow函数，接下来以此类推。这些活动记录将不断堆积，直到堆栈的空间被用完。在我的系统中，当系统崩溃时，count大约为4900。我所使用的开发环境Visual Studio默认分配1MB的堆栈空间，意味着每次调用这个函数时，即使没有任何局部变量或参数，仍然会创建一条超过200字节的活动记录。

4.4.3 生命期

变量的生命期是指从它的分配直到销毁之间的时期。对于在堆栈上所分配的变量（即局部变量或形式参数），对生命期的处理是隐式进行的。变量是在函数被调用时分配的，并在函数结束时被销毁。对于在堆上分配的变量（即用new动态分配的变量），它的生命期是由我们自己掌握的。管理动态分配变量的生命期是每个C++程序员的基本素养。最明显的问题是可怕的内存泄漏，即在堆上分配了内存但没有被销毁并且没有被任何指针所引用。以下是一个简单的例子：

```
❶ int *intPtr = new int;
❷ intPtr = NULL;
```

在这段代码中，我们声明了一个指向整数类型的指针❶，并为它分配位于堆上的一个整数，以对它进行初始化。然后，在第二行代码中，我们把这个整型指针设置为 NULL❷（它只是零这个数字的别名）。但是，用 new 所分配的那个整数仍然存在。它孤单而悄无声息地在堆中发呆，等待永远不会到来的销毁。我们无法销毁这个整数，因为当销毁一块内存的时候，需要在调用 delete 时加上指向这块内存的指针，但此时已经不存在指向这块内存的指针了。如果我们在上面这段代码之后加上 delete intPtr 这条语句，将会得到一个错误，因为此时 intPtr 已经是零。

有时候，我们遇到的问题并不是永远不会被销毁的内存，而是正好相反：两次销毁同一块内存。而这将产生一个运行时错误。这看上去像是一个很容易避免的问题：只要不对同一个变量两次调用 delete 就可以了。但是，当多个变量指向同一块内存的时候，情况就变得复杂了。如果多个变量指向同一块内存并且调用 delete 删除了其中任意一个变量，实际上就为所有的变量销毁了这块内存。如果我们没有明确地把这些变量设置为 NULL，它们此时就成为了野引用（dangling reference），对野引用调用 delete 将会产生运行时错误。

4.5 解决指针问题

现在，读者可能已经准备好了用指针来解决一些问题，那么让我们观察其中的一些问题并讨论怎样使用指针和动态内存分配来解决它们。首先，我们将使用一些动态分配的数组，演示怎样通过自己的操作对堆内存进行追踪。然后，我们再讨论一个真正的动态结构。

4.5.1 可变长度的字符串

在第一个问题中，我们将创建函数对字符串进行操作。这里所说的字符串就是最基本意义上的字符串：一个字符序列，不管其中存储了多少个字符。假设需要支持 3 个函数对这种字符串进行操作。

可变长度的字符串操作

为下面这 3 个字符串函数编写基于堆的实现：

- append：这个函数接受 1 个字符串和 1 个字符为参数，并把这个字符追加到这个字符串的尾部。

- concatenate: 这个函数接受 2 个字符串，并把第 2 个字符串的所有字符追加到第 1 个字符串的尾部。
- characterAt: 这个函数接受 1 个字符串和 1 个数字为参数，并返回字符串中这个数字所指定位置的字符（字符串的第 1 个位置的编号为 0）。

编写代码，假设 characterAt 函数会被频繁调用，而另两个函数被调次的次数则相对少一些。这些操作的相对效率应该反映调用的频率。

在这个例子中，需要选择字符串的表现形式，允许快速高效的 characterAt 函数。这意味着我们需要一种快速的方式查找一个特定的字符。读者可能记得我们在前一章讨论过数组对于这个任务是最为擅长的，因为它支持随机访问。因此，让我们用字符数组来解决这个问题。apend 和 concatenate 函数需要改变字符串的长度，这意味着我们会遇到以前所讨论过的所有数组问题。由于在这个问题中，对于字符串的长度并没有内在的限制，因此无法为数组选择一个较大的初始长度并期望它最符合要求。反之，我们需要在运行时改变数组的长度。

首先，我们为字符串类型创建一个 typedef 声明。由于需要动态创建数组，因此把这个字符串类型声明为 char 类型的指针。

```
typedef char * arrayString;
```

在此基础上，我们开始编写函数。我们所采用的原则是首先从自己掌握的知识开始，这样很快就可以完成 characterAt 函数：

```
char characterAt(arrayString s, int position) {
    ❶return s[position];
}
```

在第 3 章中，我们知道如果一个指针被赋值为一个数组的地址，就可以使用常规的数组记法来访问这个数组中的元素❶。但是，如果 position 实际上并不是数组 s 的合法元素编号，就会出现不良的结果。这段代码把验证第 2 个参数的合法性的责任放在调用者的肩上。在本章的习题中，我们将考虑一种替代方案。现在，我们暂且把目光转移到 append 函数上。我们可以想象这个函数一般将完成什么任务，但是为了保证所有的细节都落到实处，还是观察一个例子比较妥当。我把这种技巧称为通过实例来解决问题。

首先是函数或程序的实质性示例输入。写下与输入有关的所有细节，同时涵括与输出

有关的所有细节。接着，在编写代码的时候，我们将为基本情况编写代码，同时进行双重检查，观察每个步骤是怎样对示例进行转换的，以保证达到所需要的输出状态。在处理指针和动态分配的内存时，这种技巧尤其适用，因为程序中所发生的很多事情都不是直接能够观察得到的。接下来，在纸上画出一个例子，迫使自己追踪内存中所有发生变化的值，不仅仅是那些直接由变量所表示的值，还包括那些在堆中出现的值。

假设我们从字符串测试开始。我们在堆中有一个字符数组，其中按顺序包括了字符 t、e、s 和 t。我们想用自己的 append 函数在它的后面追加一个感叹号。图 4.3 显示了这个操作之前（a）和之后（b）的内存状态。在这两张图中，垂直的虚线左边的所有东西都位于堆栈内存中（局部变量和形式参数），这条虚线右边的所有东西位于堆内存中，也就是用 new 进行动态内存分配的。

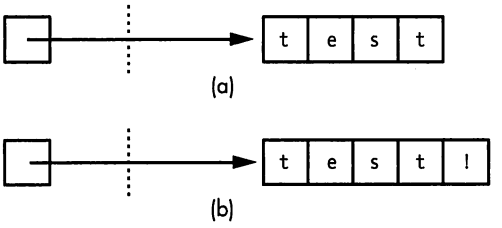

图 4.3 append 函数预想的“之前”（a）和“之后”（b）状态

观察这张图，马上就能发现我们的函数存在一个潜在的问题。根据我们对字符串的实现方式，这个函数将创建一个新数组，其长度比原来的数组大 1，并把第 1 个数组中的所有字符都复制到第 2 个数组中。但是，我们怎么才能知道第 1 个数组的长度呢？根据前一章的讨论，我们知道必须自己追踪数组的长度。因此，这段代码还少了一些东西。

如果我们已经有过标准 C/C++库的字符串的使用经验，可能已经知道缺少的部分是什么。但是如果不知道，也可以很快找出来。记得我们所采用的问题解决技巧之一就是寻找类比。也许我们应该考虑其他关于长度未知的问题。回到第 2 章，我们处理了“Luhn 检验和验证”问题中任意数字数量的标识号。在那个问题中，我们并不知道用户将输入多少个数字。最后，我们编写了一个 while 循环，一直迭代到最后一个所读取的字符是行末符。

遗憾的是，这个数组的最后并不存在行末符。但是，如果在所有的字符串数组后面添加一个行末符会怎么样呢？这样就可以像确定标识号的位数一样确定数组的长度了。这种方法的唯一缺点是无法在字符串内部使用行末符了，除非把它当作字符串终止符。根据字符串的使用方式，这个限制可能并不重要。但是，为了获得最大限度的灵活性，最好选择一个绝对不会与实际期望使用的值混淆的字符。因此，我们用 0 来终结字符串，因为 0 在 ASCII 码和其他字符码系统中表示 null 字符，这也是标准 C/C++库所使用的方法。

弄清了这个问题之后，我们深入讨论 append 函数对示例数据所执行的具体操作。这个函数将接受 2 个参数，第 1 个参数是个 arrayString，它是个指向堆中一个字符数组的指针。第 2 个参数是需要被追加的字符。为了保持简明，我们首先编写 append 函数的框架以及对它进行测试的代码，如下所示：

```
void append(❶arrayString& s, char c) {
}
void appendTester() {
    ❷arrayString a = new char[5];
    ❸a[0] = 't'; a[1] = 'e'; a[2] = 's'; a[3] = 't'; a[4] =  0;
    ❹append(a, '!');
    ❺cout << a << "\n";
}
```

appendTester 函数在堆上为字符串分配内存❷。注意这个数组的长度是 5，因为除了 test 这个单词的 4 个字母之外，还需要一个空间存储表示终止符的 null 字符❸。接着，我们调用 append 函数❹，此时它只是个空壳。在编写这个空壳的时候，我意识到 arrayString 参数必须是个引用（&）❶，因为这个函数将在堆上创建一个新数组。总之，这正是使用动态内存的要点所在：当字符串改变大小时创建一个新数组。因此，变量 a 在传递给 append 时的值并不是它在这个函数执行期间的值，因为它需要指向一个新数组。注意，由于我们像标准库一样使用了 null 字符终止符，因此可以把指针 a 所引用的数组直接发送给输出流以检查它的值❺。

图 4.4 显示了我们对这个函数测试用例所执行的操作的新理解。数组终止符已经存在，为了清晰起见用 NULL 表示。在图 4.4（b）状态之后，很显然 s 指向一块新分配的内存。以前的数组现在位于一个阴影框中。在这两张图中，我使用阴影框表示已经被销毁的内存。在图中包含已经分配的内存可以提醒我们实际执行的销毁工作。

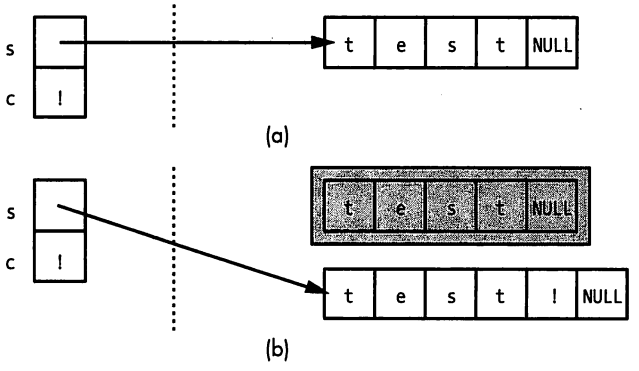

图 4.4 append 函数执行之前（a）和之后（b）经过更新的详尽内存状态

一切就绪之后，我们就可以编写这个函数了：

```
void append(arrayString& s, char c) {
    int oldLength = 0;
  ❶while (s[oldLength] != 0) {
        oldLength++;
    }
  ❷arrayString newS = new char[oldLength + 2];
  ❸for (int i = 0; i < oldLength; i++) {
        newS[i] = s[i];
    }
  ❹newS[oldLength] = c;
  ❺newS[oldLength + 1] = 0;
  ❻delete[] s;
  ❼s = newS;
}
```

这段代码需要解释的地方很多，我们将逐段对它进行分析。在函数一开始，我们用一个循环查找表示数组终止符的 null 字符❶。当这个循环结束时，变量 oldLength 的值是数组中合法字符的数量（即不包括表示终止的 null 字符）。我们在堆上分配一个长度为 oldLength + 2 的新数组❷。如果光凭头脑想象，这是很容易搞错的细节之一，但是如果用图的形式表示，一切就变得显而易见。对测试例执行这段代码的结果如图 4.5 所示，我们看到变量 oldLength 的值是 4。oldLength 之所以为 4 是因为 test 具有 4 个字符。图 4.5（b）部分的新数组需要 6 个字符，因为需要为被追加字符和 null 终止符留出空间。

分配了新数组之后，我们把所有的合法字符从旧数组复制到新数组❸，然后把需要追加的字符❹和 null 终止符❺赋值到新数组中适当的位置。同样，这张图可以帮助我们理清头绪。为了更加清晰，图 4.5 显示了变量 oldLength 的值是怎样计算产生的，并且显示了这个值所表示的是新数组中的哪个位置。有了这些视觉线索之后，就很容易知道如何在两条赋值语句中使用正确的下标了。

append 函数的最后三行代码与图 4.5 中（b）部分的阴影框有关。为了避免内存泄漏，必须销毁参数 s 原先在堆中所指向的数组❻。最后，我们让 s 指向这个新的、更长的数组❼。需要注意的是，内存泄漏在 C++编程中经常发生的原因之一是：只有当被泄漏的内存总量相当庞大时，程序和整体系统才会显示不良的后果。因此，在测试期间，程序员很可能没有注意到内存泄漏。所以，作为程序员，我们必须小心谨慎，始终要记得考虑堆内存分配的生命期。每当我们使用关键字 new 时，要考虑在哪里以及在什么时候使用对应的 delete。

注意这个函数是怎样与这张图紧密对应的。有了清晰的图，很多编程陷阱就会变得一

目了然，我希望更多的程序员新手能够在编写代码之前首先画图。这就回到了最基本的问题解决原则：始终要制定计划。对于一个问题而言，一张结构良好的图就像开车进行长途度假之前设置好导航一样。在一开始的时候稍微多花些精力，在以后可以避免更多的精力和挫折。

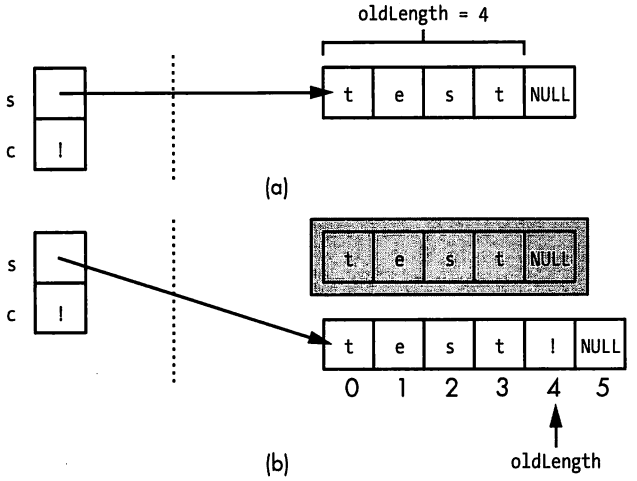

图 4.5 在 append 函数执行“之前（a）”和“之后（b）”显示局部变量、参数和被分配的内存之间的关系

建图

绘图所需的工具只有纸和笔。但是如果有时间，我推荐使用绘图程序。有一些专门为编程问题而设计的包含模板的绘图工具，但任何基于向量的绘图程序都可以满足要求（向量这个术语在这里表示用直线和曲线进行操作的程序，而不是像 Photoshop 那样的像素绘图程序）。对于本书的插图，我是用一种叫 Inkscape 的免费程序完成的。在计算机上绘图可以使我们有组织地把图放在与图所描述的代码相同的地方。过了一段时间重新回顾时，图可以使代码更清晰、更容易理解。最后，复制和修改由计算机所创建的图是非常容易的，我在根据图 4.4 创建图 4.5 时就是这样做的。如果还需要添加一些简单的临时注释，可以打印出一份拷贝再在上面进行操作。回到我们的 append 函数，代码看上去很健壮，但是要记住这些代码是建立在一个特定实例的基础之上的。因此，我们不应自大，以为这段代码可以适用于所有合法的情况。具体地说，我们需要检查特殊情况。在编程中，特殊情况是指合法的数据导致正确的代码流产生错误结果的情况。

注意 这个问题与不良数据（例如超出范围的数据）不同。在本书的代码中，我们假设程序或单独函数的输入数据是没有问题的。例如，如果程序期待接受一系列由逗号分隔的整数，那么我们就假设这个程序不会接收无关的字符，例如非数字字符等。这种假设可以合理地控制代码的长度，避免不断重复相同的数据检查代码。但是，在现实世界中，应该采取合理的预警措施防止不良输入，这就是程序的健壮性。健壮的程序对于不良的输入也会表现出良好的行为。例如，这种程序在遇到不良输入时会向用户提示一条错误消息而不是直接崩溃。

检查特殊情况

我们再次观察 append 函数，对特殊情况进行检查。也就是说，确保所有可能出现的良好输入值之中不会有任何罕见的情况。最常见的特殊情况是极端值，例如可能出现的最小或最大的输入。对于 append 函数而言，字符串数组不存在最大长度，但是存在最小长度。如果字符串不包含任何合法的字符，它实际上将对应一个只包含 1 个字符的数组（这个字符是 null 字符）。和前面一样，我们通过绘图使情况变得更直观。假设我们向一个空字符串添加了一个感叹号，如图 4.6 所示。

当我们观察这张图时，它看上去并不像是一种特殊情况，但是我们应该针对这种情况运行这个函数并检查结果。我们向 appendTester 代码中添加下面这几行：

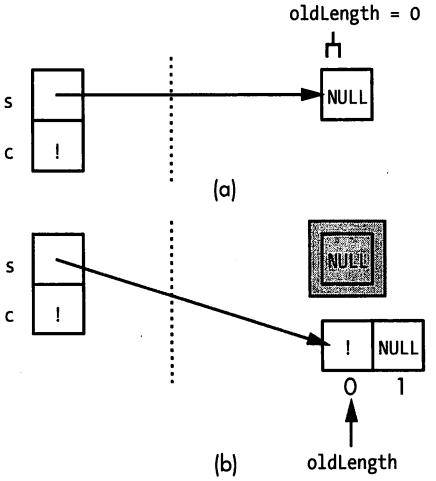

图 4.6 测试 append 函数的最小情况

```
arrayString b = new char[1];
b[0] = 0;
append(b, '!');
cout << b << "\n";
```

这种做法也是可行的。现在可以合理地保证 append 函数是正确的，但我们是否喜欢这种方法呢？代码看上去简单明了，并且没有任何怪异之处，但是对于一个简单的操作而言，代码看上去有点冗长了。当我提前思考 concatenate 函数时，突然想到它和 append 函数一样，也需要确定一个字符串数组的长度，甚至是两个字符串数组的长度。由于这两个操作都需要一个循环来寻找表示终结字符串的 null 字符，我们可以把这段代码放在一个单独的函数中，可以在 append 和 concatenate 函数中根据需要调用这个函数。让我们试着采用这种做法，并对 append 函数进行相应的修改。

```
int length(arrayString s) {
  ❶int count = 0;
    while (s[count] != 0) {
        count++;
    }
    return count;
}
void append(arrayString& s, char c) {
  ❷int oldLength = length(s);
    arrayString newS = new char[oldLength + 2];
    for (int i = 0; i < oldLength; i++) {
        newS[i] = s[i];
    }
    newS[oldLength] = c;
    newS[oldLength + 1] = 0;
    delete[] s;
    s = newS;
}
```

length 函数❶中的代码在本质上与前面 append 函数开始时的那段代码相同。在 append 函数本身之中，我们用一个对 length 函数的调用代替了那段代码❷。length 函数称为帮助函数，这种函数封装了某种操作，可以被其他函数所调用。除了减少代码的长度之外，消除冗余代码也意味着代码更可靠、更容易修改。它还可以帮助我们解决问题，因为帮助函数把代码划分为更小的块，使我们更容易认识到代码复用的机会。

复制动态分配的字符串

现在可以处理 concatenate 函数了。我们采取了与 append 函数相同的方法。首先，我们为这个函数编写一个空壳，确定它的参数以及参数的类型。接着，我们绘制一个测试用例的图，

最后编写代码以匹配这张图。以下是这个函数的壳，同时包括了一些额外的测试代码：

```
void concatenate(❶arrayString& s1, ❷arrayString s2) {
}
void concatenateTester() {
    arrayString a = new char[5];
    a[0] = 't'; a[1] = 'e'; a[2] = 's'; a[3] = 't'; a[4] =  0;
    arrayString b = new char[4];
    b[0] = 'b'; b[1] = 'e'; b[2] = 'd'; b[3] = 0;
    concatenate(a, b);
}
```

记住，对这个函数的描述说明了第 2 个字符串（第 2 个参数）中的字符被追加到第 1 个字符串的尾部。因此，concatenate 函数的第 1 个参数将是一个引用参数❶，原因与 append 函数的第 1 个参数相同。但是，第 2 个参数❷不应该被这个函数所修改，因此它将是个值参数。对于这个测试用例而言，我们是把字符串 bed 追加到字符串 test 的尾部。图 4.7 显示了这个操作之前和之后的状态。

这张图的细节与 append 函数的图非常相似。对于 concatenate 函数，我们首先在堆上分配 2 个动态分配的数组，分别由 2 个参数 s1 和 s2 所指向。当这个函数结束时，s1 将指向堆上一个长度为 9 个字符的新数组。s1 原先所指向的数组已经被销毁。s2 以及它所指向的数组保持不变。虽然在图中包含 s2 以及它所指向的 bed 数组似乎没什么意义，但是在试图避免编码错误时，追踪哪些东西不会发生变化也是非常重要的，这并不亚于追踪那些会发生变化的东西。和 append 函数的图一样，我对旧数组和新数组中的元素进行了编号。现在一切已经就绪，可以编写这个函数的代码了。

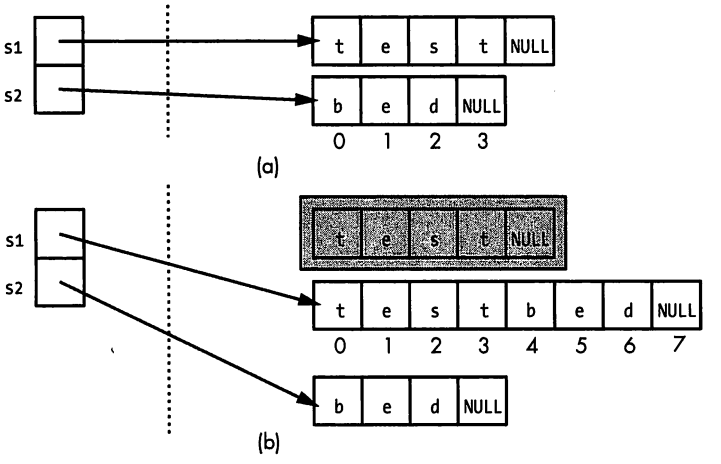

图 4.7 显示了 concatenate 方法执行“之前”（a）和“之后”（b）的状态

```
void concatenate(arrayString& s1, arrayString s2) {
  ❶int s1_OldLength = length(s1);
    int s2_Length = length(s2);
    int s1_NewLength = s1_OldLength + s2_Length;
  ❷arrayString newS = new char[s1_NewLength + 1];
  ❸for(int i = 0; i < s1_OldLength; i++) {
        newS[i] = s1[i];
    }
    for(int i = 0; i < s2_Length; i++) {
        newS[❹s1_OldLength + i] = s2[i];
    }
  ❺newS[s1_NewLength] = 0;
  ❻delete[] s1;
  ❼s1 = newS;
}
```

首先，我们确定需要连接的2个字符串的长度❶，然后把这2个值相加得到连接后的字符串的长度。注意这些长度值只包括合法字符的数量，并不包括 null 终止符。因此，当我们在堆上创建数组以存储这个新字符串时❷，所分配的空间要比合并后的长度大1，以容纳表示终止字符串的 null 字符。接着，我们把原先2个字符串中的字符复制到这个新字符串中❸。第1个循环非常简单，但是要注意第2个循环中对下标的计算❹。我们把 s2 从开始位置复制到 newS 的中间，这是从一个范围的值转换为另一个范围的值的例子，我们在第2章已经进行过这样的操作。通过观察图中的元素编号，可以发现需要把哪些变量放在一起以计算正确的下标值。这个函数的剩余部分把 null 终止符放在新字符串的尾部❺。和 append 函数一样，我们销毁第1个参数所指向的原先那块堆内存❻，并使第1个参数指向新分配的字符串❼。

这段代码看上去是可行的，但是和以前一样，我们需要保证这个函数不仅适用于测试用例，并且对于所有可能出现的情况都能应付自如。最可能造成麻烦的情况是其中一个字符串（或两者）的长度为零的时候（只包含 null 终止符）。在进行下一步工作之前应该明确地对这些情况进行检查。注意，当我们检查使用指针的代码的正确性时，还应该观察指针本身，而不仅仅是它们在堆中所引用的值。下面是一个测试用例：

```
arrayString a = new char[5];
a[0] = 't'; a[1] = 'e'; a[2] = 's'; a[3] = 't'; a[4] =  0;
arrayString c = new char[1];
c[0] = 0;
concatenate(c, a);
cout << a << "\n" << c << "\n";
❶ cout << (void *) a << "\n" << (void *) c << "\n";
```

我想要保证调用 concatenate 函数后 a 和 c 都指向 test 字符串，即它们都指向具有相同值的数组。但是，同样重要的是它们指向不同的字符串，如图 4.8（a）所示。我通过第 2 条输出语句检查这个情况，方法是把这条语句中的变量的类型修改为 void *，使输出流显示指针的原始值❶。如果两个指针本身具有相同的值，那么我们可以认为这两个指针形成了交叉链接，如图 4.8（b）所示。当指针不经意间形成了交叉链接之后，就会出现微妙的问题，因为更改堆中一个变量的内容会神秘地更改另一个变量的内容，它们实际上是同一个变量。但是在大型的程序中，很难发现这一点。另外，记住如果两个指针形成了交叉链接，当其中一个指针通过 delete 被销毁时，另一个指针就变成了野引用。因此，我们在检查代码的时候应该小心谨慎，需要认真检查潜在的交叉链接的情况。

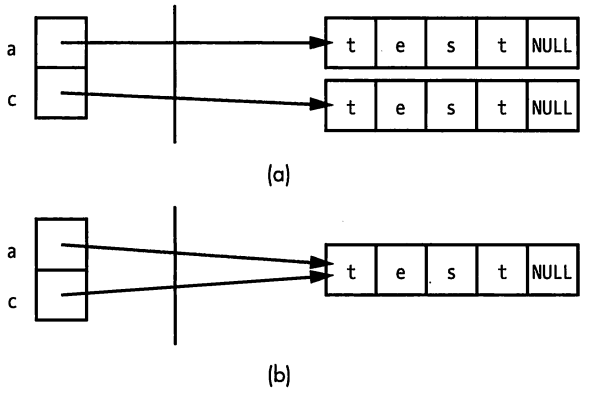

图 4.8 concatenate 函数应该产生两个不同的字符串（a），而不是两个交叉链表的指针（b）

实现了全部 3 个函数（characterAt、append 和 concatenate）之后，这个程序就完成了。

4.5.2 链表

现在我们将讨论一些更为复杂的东西。指针操作实际上是非常复杂的，为了保持简单，我们要学会怎样绘图。

追踪未知数量的学生记录

在这个问题中，我们将编写函数以存储和操纵一个学生记录的集合。学生记录包含了学生编号和成绩，它们都是整数。我们需要实现下面这些函数：

- addRecord: 这个函数接受一个指向学生记录（包括学生编号和成

绩）集合的指针为参数，并向这个数据集合添加一条新记录。

- averageRecord：这个函数接受一个指向学生记录集合的指针为参数，并以 double 值的形式返回这个集合中学生的平均成绩。

这个集合的长度是任意的。addRecord 操作预计会被频繁调用，因此它必须高效地实现。

有一些方法可以符合上面的要求，但是我们打算选择一种能够帮助自己练习基于指针的问题解决技巧的方法：链表。读者以前可能见过链表，但是对于从未接触过链表的人而言，链表的引入就像航海家的前方出现了一片全新的宽阔大海。以前提到的所有解决方案，对于优秀的问题解决者而言，只要有足够的时间和缜密的思路，都有能力独立完成。但是，大多数程序员在没有任何帮助的情况下很难透彻地理解链表的概念。不过，一旦理解了它的基础特性之后，就能够接着理解其他的链式结构，从而极大地拓展解决问题的能力。链表是一种真正的动态结构。我们的字符串数组存储在动态分配的内存中，但一经创建之后，它们就成了静态的结构，不能增长或缩小，只能够被替换。反之，链表就像菊花链一样随时可以增长或缩小。

创建节点列表

让我们创建一个包含学生记录的链表实例。为了创建链表，需要一个结构，除了包含链表所表示的集合所存储的数据之外，还需要包含一个指向相同结构类型的指针。对于我们所处理的问题，这个结构将包含一个学生编号和成绩。

```
struct ❶listNode {
  ❷int studentNum;
    int grade;
  ❸listNode * next;
};
❹ typedef listNode * studentCollection;
```

这个结构的名称是 listNode❶。用于创建链表的结构总是被称为节点。这个名称类似于植物学术语，表示茎干上发出新枝的地方。节点包含了组成真正的节点所“负载”的学生编号❷和成绩。节点还包含了一个指向我们所定义的结构的指针❸。大多数程序员第一次看到这样的指针时都会感到有点困惑，甚至觉得它在语义上是行不通的：怎么能根据自身定义一个结构呢？但这是合法的，其含义很快就会变得十分清晰。注意，节点中的自引用指针一般取 next、nextPtr 之类的名称。最后，这段代码为这种节点类型声明了一个 typedef❹，

它可以提高函数的可读性。现在，我们就可以利用这些类型创建一个示例链表了：

```
❶ studentCollection sc;
❷ listNode * node1 = new listNode;
❸ node1->studentNum = 1001; node1->grade = 78;
   listNode * node2 = new listNode;
   node2->studentNum = 1012; node2->grade = 93;
   listNode * node3 = new listNode;
❹ node3->studentNum = 1076; node3->grade = 85;
❺ sc = node1;
❻ node1->next = node2;
❼ node2->next = node3;
❽ node3->next = NULL;
❾ node1 = node2 = node3 = NULL;
```

我们首先声明了一个 studentCollection 结构 sc❶，它最终将成为我们所创建的链表的名称。接着，我们声明了 node1❷，这是个指向 listNode 的指针。同样，studentCollection 是 node*的同义词，但是为了提高可读性，我只把 studentCollection 类型用于表示整个链式结构的变量。在声明了 node1 指针并使它指向堆中一个新分配的 listNode❷之后，我们对这个节点中的 studentNum 和 grade 字段进行赋值❸。此时，next 字段尚未赋值。虽然本书并不是讨论语法的书籍，但是如果读者之前未曾见到过->记法，只要知道它表示一个由指针所指向的结构（或类）的字段就可以了。因此，node1->studentNum 表示“node1 所指向的结构的 studentNum 字段”，相当于(*node1).studentNum。接着，我们为 node2 和 node3 指针重复这个过程。对最后一个节点进行了赋值之后，内存的状态就如图 4.9 所示。在这些图中，我们将使用以前对数组所使用的分格方框记法来显示节点结构。

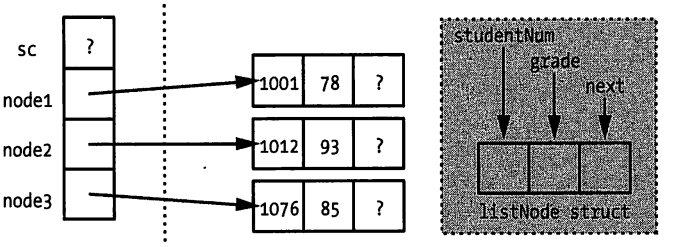

图 4.9 创建一个示例链表，完成一半时的情况

创建了所有的节点之后，现在我们可以把它们串在一起形成一个链表。这正是前面的代码清单的剩余部分所完成的任务。首先，我们使 studentCollection 变量指向第 1 个节点❺，接着让第 1 个节点的 next 字段指向第 2 个节点❻，然后让第 2 个节点的 next 字段指向第 3 个节点❼。在接下来的步骤中，我们把 NULL（它同样是零的同义词）赋值给第 3 个节点的 next 字段❽。这种做法的意图与前一个问题中我们在数组最后添加一个 null 字符串的意图相

同，也就是表示这个结构的结束。就像需要一个特定的字符表示数组的结尾一样，我们需要把链表的最后一个节点的 next 字段设置为零值，以便知道它是最后一个节点。最后，为了理清思路并防止可能出现的交叉链接问题，我们把每个单独的节点指针都设置为 NULL❾。图 4.10 显示了内存的最终状态。

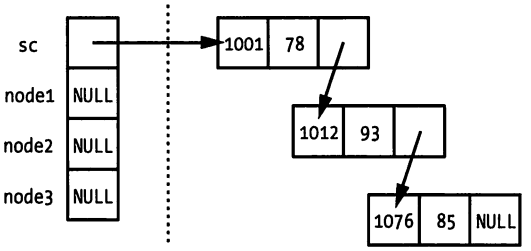

图 4.10 完成后的示例链表

观察了它的可视化表现形式之后，我们就很清楚这个结构为什么称为链表了：列表中的每个节点都被链接到下一个节点。我们常常看到线性绘制的链表，但作者更倾向于这张图中所显示的散布在内存中的外观，因为它强调了这些节点除了链接之外，彼此之间再没有什么关系。每个节点可以位于堆中的任意位置。读者可以仔细追踪这段代码，直到理解了它与这张图的吻合原因。

注意 在结束状态时，只有一个基于堆栈的指针处于使用状态，也就是指向第 1 个节点的 studentCollection 变量 sc。指向链表的第 1 个节点的外部指针称为头指针。在符号层次上，这个变量表示整体的链表，但它直接所引用的当然只是第 1 个节点。为了访问第 2 个节点，必须通过第 1 个节点。为了访问第 3 个节点，必须通过前两个节点，接下来以此类推。这意味着链表只提供了线性访问，而不是像数组一样提供随机访问。线性访问是链表结构的弱点。链表结构的优点正如之前所说的那样，能够通过添加和删除节点实现结构的增长或收缩，而不需要创建一个全新的结构并复制原先的数据。对于数组，则只能采用后面这种方法。

向链表添加节点

现在让我们实现 addRecord 函数。这个函数将创建一个新节点并把它连接到一个现有的链表上。我们将使用与前一个问题相同的技巧，首先创建一个函数空壳和一个示例调用。为了进行测试，我们将向前面的代码清单中添加代码，因此变量 sc 已经作为指向包含 3 个节点的链表的指针存在。

```
void addRecord(studentCollection& sc, int stuNum, int gr) {
}
❶ addRecord(sc, 1274, 91);
```

同样，这个调用❶将出现在前面这个代码清单的最后。编写了包含参数的函数空壳之后，我们可以画出这次调用“之前”的状态，如图 4.11 所示。

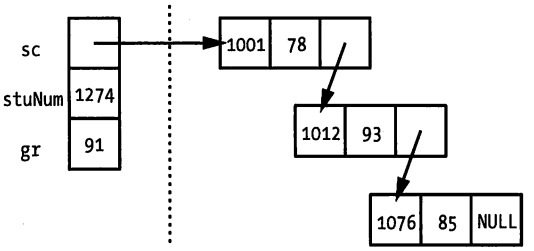

图 4.11 addRecord 函数执行“之前”的状态

但是，对于“之后”状态，我们可以进行选择。我们能够猜到需要在堆中创建一个新节点并把参数 stuNum 和 gr 的值复制给新节点的 studentNum 和 grade 字段。问题在于，从逻辑上说这个节点应该出现在链表中的哪个位置呢？最显而易见的选择是让它出现在链表的最后，把那个 next 字段为 NULL 的节点的 next 字段指向这个新节点。这种做法如图 4.12 所示。

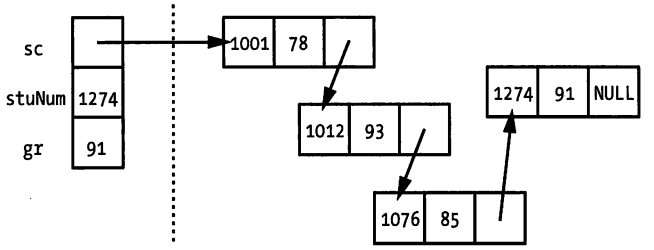

图 4.12 addRecord 函数执行“之后”的状态

但是，如果我们可以假设记录的顺序无关紧要（并不需要保证与它添加到集合中的原先顺序相同），那么上面这种做法就是错误的选择。为什么呢？我们可以考虑一个包含了 3000 条记录而不是 3 条记录的集合。为了到达链表中最后一条记录以修改它的 next 字段，需要遍历所有 3000 个节点。这是无法接受的低效率，因为我们可以把这个新节点插入到链表中而不需要遍历任何现有的节点。

如图 4.13 所示，在创建了新节点之后，把它链接到链表的起始处而不是链接到尾部。在“之后”状态中，我们的头指针 sc 指向这个新节点，而这个新节点的 next 字段则指向原

先为链表第 1 个节点的那个节点，也就是学生编号为 1001 的节点。注意，当我们为新节点的 next 字段赋值时，唯一发生变化的现有指针是 sc，剩余节点的任何值都没有发生变化，甚至未曾被读取。通过我们所创建的图，可以编写相应的代码：

```
void addRecord(studentCollection& sc, int stuNum, int gr) {
  ❶listNode * newNode = new listNode;
  ❷newNode->studentNum = stuNum;
    newNode->grade = gr;
  ❸newNode->next = sc;
  ❹sc = newNode;
}
```

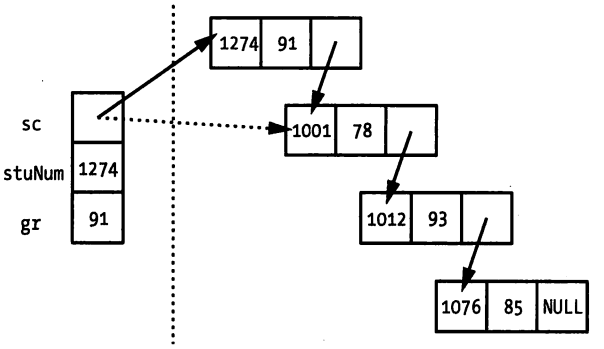

图 4.13 addRecord 函数可接受的“之后”状态。虚线箭头表示 sc 指针原先所存储的值

同样，我们强调把图转换为代码要比在头脑里凭空想象容易得多。这段代码直接来自上面的示意图。我们创建了一个新节点❶，并根据参数对学生编号和成绩进行赋值❷。接着，我们把这个新节点链接到链表中，首先把新节点的 next 字段指向以前的第 1 个节点（将它赋值为 sc 的值）❸，然后让指针 sc 指向这个新节点本身❹。注意，最后两个步骤必须按上面的顺序进行。我们需要使用 sc 指针原先的值，然后才能修改它。另外，注意由于我们修改了 sc 指针，所以它必须是个引用参数。

和往常一样，当我们根据一个示例创建代码时，必须检查潜在的特殊情况。在本例中，这意味着该函数必须能够适用于空链表。在我们的字符串数组中，空字符串仍然是个合法的指针，因为我们仍然有一个需要指向的数组，只不过它是个只包含了 null 终止符的数组。但是，当前节点的数量与记录的数量相同，空链表将是一个 NULL 头指针。当头指针为 NULL 时，如果我们尝试把示例数据插入到这个链表中会不会有问题呢？图 4.14 显示了链表“之前”状态和我们希望出现的“之后”状态。

按照这种情况模拟代码的运行，我们发现它能够很好地处理此种情况。新节点仍然和

以前一样被创建。由于 sc 在“之前”状态下是个 NULL 指针，因此当这个值❸被复制到新节点的 next 字段时，它正是我们期望出现的行为，并且这个单节点链表被正确地结束。注意，如果我们采用了另一种实现思路，也就是把新节点添加到链表的尾部而不是头部，初始链表为空就是个需要特殊处理的情况，因为这是唯一需要对 sc 指针进行修改的情况。

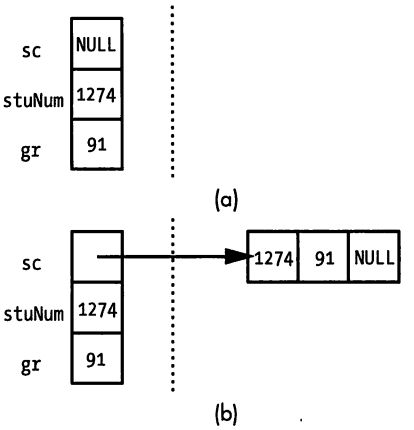

图 4.14 链表长度最小情况下 addRecord 函数执行“之前”（a）和“之后”（b）的状态

链表的遍历

现在应该处理 averageRecord 函数了。和之前一样，我们首先创建一个函数空壳和一张图。下面是函数空壳和示例调用。假设这个示例调用❶发生在原先的示例链表被创建之后，如图 4.10 所示。

```
double averageRecord(studentCollection sc) {
}
❶ int avg = averageRecord(sc);
```

正如我们所看到的那样，和前一章计算数组的平均值一样，我选择了用 int 类型计算平均值。但是，取决于具体的问题，用浮点数来计算平均值可能更为合适。现在我们需要一张图，但图 4.9 已经很好地描绘了“之前”状态。我们并不需要一张图表示“之后”状态，因为该函数并不打算对这个动态数据结构进行修改，只是根据它的内容执行一些计算而已。我们只需要知道预期的结果，在此例中大约是 85.3333。

因此，我们实际上是怎样计算平均值的呢？根据我们计算数组平均值的经验，已经掌握了基本概念。我们需要把这个集合中的每个值加在一起，然后把得到的和除以值的个数。根据以前求数组平均值的代码，我们需要一个循环来检查每个值，这个循环计数从 0 开始，直到比数组的长度少 1，并使用循环计数器作为数组下标。但是在这里无法使用 for 循环，

因为我们预先并不知道链表中有多少个元素。我们必须一直执行迭代，直到发现某个节点的 next 字段的值为 NULL，即表示这个链表已经结束。这提示我们应该使用 while 循环，有点类似本章前面处理未知长度的数组一样。按照这种方式从头到尾查看链表的每个节点的方式称为链表的遍历，它是链表最基本的操作之一。让我们在解决这个问题时引入链表遍历的思路。

```
double averageRecord(studentCollection sc) {
  ❶int count = 0;
  ❷double sum = 0;
  ❸listNode * loopPtr = sc;
  ❹while (loopPtr != NULL) {
        ❺sum += loopPtr->grade;
        ❻count++;
        ❼loopPtr = loopPtr->next;
    }
  ❽double average = sum / count;
    return average;
}
```

我们首先声明一个变量 count，用于保存在链表中所遇到的节点的数量❶，它也可以用来表示集合中值的个数并用来计算平均值。接着，我们声明了一个变量 sum 以存储链表中当前的成绩之和❷。然后，我们声明一个称为 loopPtr 的 listNode *指针，用于对这个链表进行遍历❸。它用于处理整数的 for 循环时所使用的整数循环变量。它追踪链表中的位置，并不是通过使用位置编号，而是通过存储一个指向当前正在处理的节点的指针实现。

此时，遍历本身就开始了。遍历循环将一直持续，直到循环追踪指针到达了表示链表结束的 NULL 值❹。在这个循环的内部，我们把当前所引用的节点的 grade 字段的值加到 sum 上❺。然后把 count 的值加 1❻，并把当前节点的 next 字段赋值为循环追踪指针❼。这种做法的效果就是把遍历向后移动一个节点。这也是这段代码中有点复杂的地方，所以我们要把一切都理清楚。在图 4.15 中，我显示了节点变量是怎样随着时间而变化的。字母（a）到（d）标识了代码对示例数据的执行期间的不同时间点，显示了 loopPtr 的生命期间的不同点以及 loopPtr 的值是从哪个位置获得的。（a）点是循环刚刚开始的时候，loopPtr 被初始化为 sc 指针的值。因此，指针 loopPtr 和 sc 一样指向链表中的第 1 个节点。在循环的第 1 次迭代期间，第 1 个节点的成绩值 78 被加到 sum 中。第 1 个节点的 next 值被复制给 loopPtr，使 loopPtr 现在指向链表的第 2 个节点，这是（b）点的情况。在第 2 次迭代期间，我们把 93 加到 sum 上，并把第 2 个节点的 next 字段的值复制给 loopPtr，这是（c）点的情况。最后，在循环的第 3 次也是最后一次迭代期间，我们把 85 加到 sum 上并把第 3 个节点的 next 字段的 NULL 值赋值给 loopPtr，这是（d）点的情况。当我们再次到达 while 循环的顶部时，这

个循环即告终止，因为 loopPtr 的值现在为 NULL。由于我们每次迭代时都把 count 的值增加 1，因此它现在的值是 3。

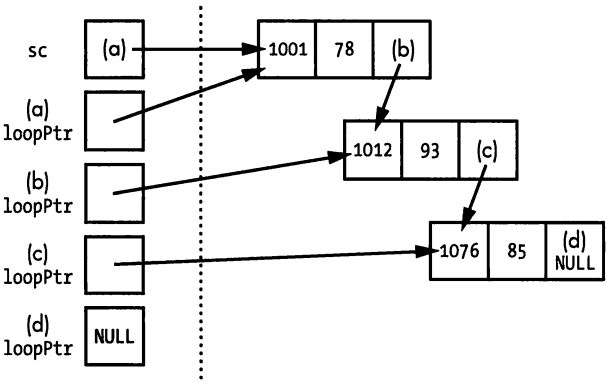

图 4.15 *局部变量 loopPtr 在 averageRecord 函数的循环迭代期间是如何变化的*

一旦循环结束之后，我们就把 sum 的值除以 count 的值，并返回其结果❽。

这段代码对于示例数据没有问题，但和往常一样，我们需要检查潜在的特殊情况。同样，对于这个链表，最明显的特殊情况是空链表。当这个函数执行时，如果指针 sc 为 NULL，会发生什么情况呢？

猜猜会怎么样？代码将会失败。（我必须举出因一个特殊情况出现不良结果的例子，不然读者很可能不把特殊情况当一回事。）处理链表的循环本身并没有什么错误。如果 sc 为 NULL，则 loopPtr 被初始化为 NULL，这个循环在刚一开始就告结束，这样 sum 变量的结果就是 0，一切都很合理。问题在于执行计算平均值的除法时❽，变量 count 的值也是 0，这意味着我们将执行除零运算，这要么导致程序崩溃，要么产生垃圾值结果。为了处理这种特殊情况，我们必须在这个函数的最后检查变量 count 的值是否为零。但是，为什么不在一开始就处理这种情况，对 sc 的值进行检查呢？让我们在 averageRecord 函数一开始时添加下面这行代码作为它新的第 1 行：

```
if (sc == NULL) return 0;
```

如上所示，处理特殊情况通常是相当简单的。我们只要花点时间确定它们的存在就可以了。

4.6 结论和未来的步骤

本章揭开了使用指针和动态内存解决问题的序幕。在本书的剩余部分，读者还会看到

大量使用指针和堆内存分配的场合。例如，我们将在第 5 章所讨论的面向对象编程技巧在处理指针方面尤其重要。它们允许我们对指针进行封装，这样就不必担心内存泄漏、野指针以及其他常见的指针陷阱。

虽然在这个领域的问题解决方面还有很多内容有待学习，但是如果读者已经掌握了本章的基本思路，已经能够自行开发基于指针结构的使用技巧以处理越来越复杂的问题了。首先，应用最基本的问题解决规则。其次，应用指针的特定规则，并使用图或类似的工具创建解决方案的视觉表现形式，然后再开始编写代码。

4.7 习题

我对于习题是极为重视的。在读完正文的内容之后，读者一定要花点时间完成下面相关的习题。

4.1 自行设计：取一个已经知道怎样用数组来解决但受限于数组长度的问题。重新编写代码，使用动态分配的数组来取消这个限制。

4.2 对于动态分配的字符串，创建一个函数 substring，它接受 3 个参数：一个 arrayString、一个表示起始位置的整数和一个表示字符长度的整数。这个函数返回一个指针，指向一块新的动态分配的字符串数组。这个字符串数组包含了原先字符串中从指定的位置开始指定长度的字符。原先的字符串不应该受到这个操作的影响。因此，如果原先的字符串是 adcdefg，位置是 3，长度是 4，则新字符串应该包含 cdef。

4.3 对于动态分配的字符串，创建一个 replaceString 函数，它接受 3 个参数，每个参数的类型都是 arrayString，分别是 source、target 和 replaceText。这个函数用 replaceText 替换 target 在 source 中的每次出现。例如，souce 指向一个包含 adcdabee 的数组，taret 指向 ab 并且 replaceText 指向 xyz，那么当这个函数执行完成之后，source 应该指向一个包含 xyzcdxyzee 的数组。

4.4 修改字符串的实现，使数组中的 location[0]存储数组的长度（因此 location[1]存储字符串中实际上的第 1 个字符），而不是使用 null 字符结尾符。实现下面这 3 个函数：append、concatenate 和 charactertAt，尽可能利用已经存储的字符串长度的信息。由于不再使用标准输出流所期待的 null 结束约定，因此必须自行编写输出函数，对它的字符串参数进行迭代，显示所有字符的内容。

4.5 编写一个 removeRecord 函数，它接受 1 个指向 studentCollection 的指针和一个学生编号为参数，从集合中删除具有指定学生编号的那条记录。

4.6 为字符串创建一种实现，使用字符链表而不是动态分配的数组。因此，我们将具有一个数据负载为单个字符的链表。这就允许字符串在不重新创建整个字符串的情况下实现增长。我们首先将实现 append 和 characterAt 函数。

4.7 在前一个习题的基础上，实现 concatenate 函数。注意如果我们调用了 concatenate(s1, s2)，并且 2 个参数都是指向各自链表的第 1 个节点的指针，这个函数应该在 s2 中为每个节点创建一份拷贝，并把它们追加到 s1 的尾部。也就是说，这个函数不应该简单地把 s1 链表的最后一个节点的 next 字段指向 s2 链表的第 1 个节点。

4.8 在字符串的链表实现中添加一个称为 removeChars 的函数，它根据位置和长度从一个字符串中删除一部分字符。例如，removeChars(s1, 5, 3)将从字符串 s1 中删除从第 5 个位置开始的 3 个字符。保证被删除的节点被正确地销毁。

4.9 想象一个节点所存储的数据不是字符而是数字的链表，每个数字是个 0～9 范围内的整数。我们可以用这种链表表示任意长度的正数。例如，149 这个数可以用一个首节点为 1、次节点为 4、第 3 个节点为 9 的链表表示。编写一个 intToList 函数，它接受 1 个整数值并生成一个这种类型的链表。提示：反向生成这个链表可能会容易得多，因此如果值为 149，首先应该创建 9 这个节点。

4.10 对于前一个习题的数字链表，编写一个函数，它接受 2 个这样的链表为参数并生成一个表示它们之和的新链表。

第 5 章

用类解决问题

在本章中，我们将讨论类和面向对象编程。和以前一样，我假设读者已经掌握了怎样在 C++中声明一个类，并了解创建一个类以及调用类的方法。我们将在下一节对这些内容进行简单的回顾，但本章的重点在于怎样用类来解决问题。

这是我觉得 C++较之其他语言存在优势之处。由于 C++是一种混合语言，C++程序员可以在适当的时候创建类，但绝不是必须创建类。反之，像 Java 或 C#这样的语言，所有的代码都必须位于类声明的内部。在编程专家手里，这不会产生什么危害，但在程序员新手那里，这可能导致坏习惯。对于 Java 或 C#程序员而言，任何东西都是对象。虽然用这些语言所编写的代码都必须封闭在对象中，但其结果并不总是能够映射到明显的面向对象设计。对象应该是有含义的，是数据以及对数据进行操作的代码的紧密结合。它不应该是囊括所有东西的垃圾收集袋。

由于我们用 C++编程，因此可以选择过程性编程和面向对象编程。我们将讨论良好的

类设计，并讨论什么时候应该使用类，什么时候不应该使用类。认识到什么情况下使用类对于达到更高层次的编程水平是极为重要的，但是认识到在什么情况下使用类反而会使情况变得更糟，也具有同等重要的意义。

5.1 类的基础知识回顾

和前几章一样，本书假设读者之前已经接触过 C++语法的基础知识，但还是对类的语法的基础知识进行回顾，以便在术语上保持一致。类是创建一个特定的代码和数据包的蓝图。根据类的蓝图所创建的每个变量称为这个类的对象。在类的外部创建并使用这个类的对象的代码称为这个类的客户。类声明对类进行命名并列出了类的所有成员（或称为项）。每个项是数据成员（在类的内部所声明的变量）或方法（在类的内部所声明的函数，又称成员函数）。成员函数可以包含一种称为构造函数的特殊类型函数，它的名称与类名相同，当这个类的对象被声明时会被隐式地调用。除了变量或函数声明的常规属性（例如类型以及函数特有的参数列表）之外，每个成员还有一个访问指示符，它表示哪些函数可以访问这个成员。公共成员可以通过类对象被任何代码来访问，包括类内部的代码、类的客户或一个子类中的代码。所谓子类，就是“继承”了一个现有类的所有代码和数据的类。私有成员只能被这个类内部的代码所访问。保护成员（我们将在本章的后面内容中讨论）与私有成员相似，区别在于子类的方法也可以引用这种成员。但是，私有成员和保护成员都不能被客户代码所访问。

与返回类型这样的属性不同，类声明中的访问指示符会一直发挥作用，直到被另一个不同的访问指示符所代替。因此，每个指示符通常只出现一次，使类的成员按照访问限制进行分组。这导致程序员常常提到类的“公共部分”或“私有部分”，例如“我们应该把这个方法放在私有部分”。

让我们观察类声明的一个小例子：

```
class ❶sample {
❷ public:
      ❸sample();
      ❹sample(int num);
      ❺int doesSomething(double param);
  private:
      ❻int intData;
  }❼;
```

这个声明首先对类进行命名❶，这样 sample 就成为一个类型名称。声明是从访问指示符

public 开始的❷，因此在到达 private 指示符❻之前，所有的东西都是公共性质的。许多程序员在类中首先包含公共声明，认为公共接口是其他读者最感兴趣的。这个例子中的公共声明是两个称为 sample 的构造函数（❸和❹）以及另一个方法 doesSomething❺。构造函数是在声明这个类的对象时被隐式地调用的。

```
sample object1;
sample object2(15);
```

这里，对象 object1 将调用第 1 个构造函数❸，即默认构造函数，它不接受任何参数。对象 Object2 将调用第 2 个构造函数❹，因为它指定了 1 个整数值，因此与第 2 个构造函数的参数签名匹配。

在类声明的最后部分是一个私有数据成员 intData❻。记住，类声明是以一个右花括号和一个分号结束的❼。这个分号看上去有点神秘，因为我们在结束函数、if 语句块或其他任何需要使用右花括号的场合都没有使用分号。这个分号的存在实际上提示类声明的同时也能表示对象声明。我们可以在右花括号和分号之间添加标识符，在创建类的同时创建这个类的对象。但是，这种做法在 C++中并不是很普遍，尤其是许多程序员习惯把类定义放在与使用它们的程序不同的文件中。在结构体声明的右花括号后面也存在这个神秘的分号。

说到结构体，读者可能知道在 C++中，结构体和类几乎表示相同的东西。它们之间唯一的区别与出现在第 1 个访问指示符之前的成员（数据或方法）有关。在结构体中，这些成员将是公共的，但它们在类中却是私有的。不过，优秀的程序员会按照不同的方式使用这两种数据结构。这就像所有的 for 循环都可以改写为 while 循环，但是优秀的程序员在简单明了的计数循环中总是使用 for 循环，以使代码更容易阅读。大多数程序员用结构体表示更为简单的数据结构，例如除了构造函数之外不再有其他方法的结构体或者这个结构体主要作为传递给另一个更大的类的方法的参数。

5.2 使用类的目的

为了认识到使用类的正确场合和错误场合，以及创建类的正确方法和错误方法，我们首先必须明确使用类的目的。在思考这个问题的时候，我们应该记住类总是可选的。也就是说，类并没有向我们提供像数组或基于指针的结构一样无法替代的功能。如果有一个程序使用一个数组存储 10000 条记录，那么就不可能在不使用数组的情况下编写同一个程序。如果有一个程序依赖链表随时增长和收缩的功能，那么就不可能在不使用链表或其他基于指针的数据

结构的情况下实现具有相同效率和相同效果的程序。但是，如果从一个面向对象的程序中把所有的类都拿走并重新编写这个程序，程序看上去将会有所不同，但程序的功能和效率却不会降低。事实上，早期的 C++编程器就像预处理器一样：C++编译器随时读取 C++源代码并输出符合传统 C 语法的新源代码，修改后的源代码再发送给 C 编译器。这就告诉我们，C++对 C 的主要扩展并不是语言的功能，而是使源代码更容易为程序员所阅读。

因此，在选择基本的类设计目标时，我们所选择的目标是帮助自己（程序员）完成任务。具体地说，由于本书所讨论的是怎么解决问题，因此应该考虑类怎样帮助我们解决问题。

5.2.1 封装

封装是指类把多块不同的数据和代码放在一个单一的包中。如果读者曾经看到过由微小颗粒所填充的胶囊，可以把它作为类比：患者服下一粒胶囊，相当于吞下了它内部包含各种独立成分的颗粒。

封装这种机制使我们在下面各节所列出的其他目标变得可行，但它本身还具有一个优点——使代码更有组织性。在一个很长的、由纯过程性代码所组成的程序清单中（在 C++中，就是只由函数所组成的代码，不包含类），很难确定函数和编译器指令的良好顺序，以至于我们很难记住它们的位置。反之，我们必须依赖开发环境来寻找函数。封装把相关的东西包装在一起。如果我们修改了一个类的方法并意识到需要观察或修改其他代码，这些代码很可能位于同一个类的其他方法中，并且很可能就在附近。

5.2.2 代码的复用

站在解决问题的角度，封装允许我们更方便地复用以前问题的代码，用它们来解决当前的问题。在很多情况下，即使我们所着手的问题与当前的项目类似，复用以前的代码仍然需要做很多工作。完整封装的类就像一个外部 USB 设备一样，只要把它插入就可以发挥作用了。但是，为了实现这个目标，必须正确地设计类，确保代码和数据被真正地封装，并尽可能不依赖于类外部的任何东西。例如，如果一个类引用了一个全局变量，那么把它复制到一个新项目时必须把这个全局变量也一并复制过去，这显然影响了类的复用性。

除了把一个程序的类复用于另一个程序之外，类还提供了一种潜力，实现更直接形式的代码复用：继承。记住，我们在第 4 章讨论了使用帮助函数“重构”两个或多个函数之间的公共代码。继承进一步拓展了这个思路。使用继承，可以创建父类，它的方法对于两个或更多个子类都是通用的，使“重构”不仅仅是几行代码，而是整个方法。继承本身就

是一个巨大的主题，我们将在本章的后面内容中讨论这种形式的代码复用。

5.2.3 问题的细分

我们反复使用的一个技巧是把一个复杂的问题细分为几个更小、更容易管理的片段。类可以很方便地把程序划分为几个功能单元。封装不仅把数据和类放在一个可复用的包中，它还把数据和代码与程序的其余部分隔离，这样我们就可以单独对这个类进行操作，而其他东西都是与之隔绝的。我们在程序中所创建的类越多，问题细分的效果就越明显。

因此，只要有可能，我们应该把类作为细分复杂问题的方法。如果类的设计良好，它可以实现功能分离，使问题更容易解决。作为附加效果，为一个问题所创建的类可以复用于其他问题，即使我们在创建它们的时候并没有考虑到这种可能性。

5.2.4 信息隐藏

有些人把信息隐藏和封装看成是同一个概念，但是在此我们还是对它们进行区分。如本章的前面内容所述，封装是把数据和代码包装在一起。信息隐藏意味着把数据结构的接口（操作以及它们的参数的定义）与数据结构的实现（或函数中的代码）进行隔离。如果一个类在编写时把信息隐藏作为目标之一，那么在修改方法的实现时可以不对客户代码（使用这个类的代码）进行任何修改。同样，我们必须弄清接口这个术语。接口不仅仅是方法以及它们的形参列表的名称，还包括对不同的方法所执行任务的解释（可能在代码文档中描述）。当我们表示修改实现方法而不改变接口时，意思是修改类方法的工作方式，但不改变它们的功能。有些编程作者把接口看成是类与客户之间的一种隐式契约：类同意不改变现有操作的效果，客户同意按照类的接口严格地使用类，忽略任何实现细节。试想有一种可以控制所有电视机的统一遥控器，不管是旧式的真空管电视机还是使用 LCD 或等离子屏幕的电视机。先后按下 2、5 和 Enter 键，所有的屏幕都会显示 25 频道，即使不同的电视机因为底层技术的差异在实现这个功能上所使用的机制大相径庭。

在不封装的情况下实现信息隐藏是不可能的。但是，由于我们已经定义了这两个术语的明确含义，因此在不实现信息隐藏的情况下实现封装是可能的。最显而易见的情况是类的数据成员被声明为公共性质。在这样的类中，类仍然进行了封装，它是把代码和数据放在一起的包。但是，客户代码现在可以访问一个重要的类实现细节：类用于存储数据的变量以及它们的类型。即使客户代码并没有直接修改类的数据而只是对它进行检查，它仍然需要这种特定的类实现。如果修改了类的任何变量的名称或类型，直接访问这些变量的客

户代码也需要随之修改。

读者的第一个想法可能是只要把所有的数据都声明为私有数据，并花费足够的时间设计成员函数列表以及它们的形参列表使其永远不需要修改，就可以确保信息隐藏。虽然这些对于信息隐藏而言确实是必须的，但并不充分，因为信息隐藏问题可能更为微妙。记住，类同意不更改它的任何方法的功能，而不管具体是什么情况。在以前的章节中，我们必须确定一个函数将要处理的最小情况或者在遇到一种异常情况下该怎么办，例如在寻找一个数组的平均数时发现数组的长度为零时该怎么办。即使是为了适应一种异常的情况而更改方法的结果都会导致接口的改变，我们应该予以避免。这也是为什么在编程中明确考虑特殊情况如此重要的另一个原因。许多时候，当一个程序的底层技术或应用程序编程接口（API）被更新时会出现问题，在过去有些会很可靠地返回-1 的系统调用现在很可能出错，而返回一个看上去随机的、但仍然是负数的值。避免这个问题的最好办法之一是在类或方法文档中陈述特殊情况的结果。如果文档表示当一个特定的情况发生时将会返回-1 这个错误代码，就应该重新考虑让自己的方法返回同样的值。

那么，信息隐藏是怎样影响解决问题的呢？信息隐藏的原则告诉程序员在编写客户代码的时候就要把类的实现细节搁在一边。或者更广泛地说，只在编写一个特定类的内部细节的时候才关注它的实现。把实现细节暂时抛诸脑后，才能消除思维的分心，把注意力集中在解决当前问题上。

但是，把信息隐藏与问题解决相关联时，我们应该注意它的限制。有时候，实现细节并不会对客户代码产生影响。在前面的章节中，我们已经看到了一些基于数组的数据结构和一些基于指针的数据结构的优点和弱点。基于数组的数据结构允许随机访问，但是无法轻易地增长或收缩。反之，基于指针的数据结构只提供了线性访问，但是在添加或删除其中的一些元素时并不需要重新创建整个结构。因此，以基于数组的结构为基础所创建的类的特性与那些以基于指针的结构为基础所创建的类不同。

在计算机科学中，我们常常讨论抽象数据类型的概念，这是最纯粹形式的信息隐藏：一种数据类型只是由它的操作所定义的。在第 4 章，我们讨论了堆栈的概念，并描述了程序的堆栈是一块连续的内存。但是，作为抽象数据类型，堆栈可以是任何能够添加并删除单独数据项的数据类型，并且这些数据项是按照与添加相反的顺序被删除的。这称为后入先出的顺序（LIFO）。没有任何硬性规定堆栈必须是一块连续的内存，用链表创建堆栈也是完全可以的。由于连续的内存和链表具有不同的属性，因此使用某种方法实现的堆栈可能与使用其他方法实现的堆栈具有不同的属性，也许会对使用堆栈的客户产生巨大的差别。

我的观点是信息隐藏对于问题解决者而言是一个非常实用的目标，它在某种程度上允许我们细分问题，单独处理程序的不同部分。但是，我们并不能完全忽略实现细节。

5.2.5 可读性

设计良好的类可以提高使用它的程序的可读性。对象可以对应于我们所观察的现实世界，因此方法调用常常具有像日常语言这样的可读性。另外，对象之间的关系常常比简单变量之间的关系更为清晰。提高可读性可以增强解决问题的能力，因为我们在开发的时候可以更轻松地理解代码。如果旧代码更容易被沿用，也能够增强复用性。

为了最大限度地利用类的可读性的优点，我们需要思考类的方法是如何被实际使用的。方法的名称应该精心选择，以反映方法的效果的最特定含义。例如，考虑一个表示金融投资的类，它包含了一个用于计算未来值的方法。compute（计算）这个名称所传达的意思远不如 computeFutureValue。甚至选择名称的正确词性也是非常有意义的。Compute FutureValue 是个动词，而 futureValue 是个名词。观察这两个名称在下面的示例代码中的用法：

```
double FV;
❶ investment.computeFutureValue(FV, 2050);

❷ if (investment.futureValue(2050) > 10000) { ...
```

如果仔细斟酌，对于独立型的调用，前者更为合理。所谓独立型的调用，就是指函数的返回类型为 void，未来值是通过一个引用参数发送给调用者的❶。当这个调用是在一个表达式中使用时，后者更为合适，也就是把未来值作为这个函数的返回值❷。

我们将在本章的后面内容中看到具体的例子，但是最大限度地遵循可读性的指导原则，即是在编写类接口的任何部分的内容时总是把客户代码放在第一位。

5.2.6 表达能力

设计良好的类的最后一个目标是表达能力，或者更广泛地称之为可写性，就是指代码的容易编写程度。一个设计良好的类一经写成之后，可以使代码的剩余部分更容易编写，就像一个良好的函数可以使代码的编写变得更为简单一样。类有效地对语言进行了扩展，相比循环、if 语句等基本的底层特性，类是更为高级的特性。在 C++中，即使是输入和输出这样的核心功能也不是语言的内在部分，而是由一组类所提供的。需要使用输入输出功能的程序必须明确包含这些类。有了类之后，以前需要许多步骤才能完成的程序操作现在只

需要几个步骤甚至只要一个步骤就可以完成了。作为问题解决者，我们应该把这个目标放在特殊的优先级上。我们应该始终考虑："这个类怎样使这个程序的剩余部分以及未来可能使用这个类的程序更容易编写？"

5.3 创建一个简单的类

理解了使用类的目的之后，现在是时候把理论转化为实践并创建一些类了。首先，我们将开发一个类，以分阶段解决下面这个问题。

5.3.1 问题：班级花名册

设计一个（或一组）类，供一个维护班级花名册的程序所使用。对于每个学生，存储他的姓名、ID和0～100范围的期末考试成绩。这个程序允许添加或删除学生记录、显示一个特定学生的记录、通过ID查找学生、支持用数字和字母表示成绩并且可以显示整个班级的平均成绩。表5.1显示了与特定的成绩相对应的字母成绩。

表5.1 字母成绩

成绩范围	字母成绩
93–100	A
90–92	A–
87–89	B+
83–86	B
80–82	B–
77–79	C+
73–76	C
70–72	C–
67–69	D+
60–66	D
0–59	F

我们首先观察一个基本的类框架，它形成了大部分类的基础。接着，我们观察这个基本框架是怎样进行扩展的。

5.3.2 基本的类框架

探索基本的类框架的最好方法是通过一个类的示例。对于这个例子，我们打算把第 3 章的 student 结构作为起点，在它的基础上创建一个完整的类。为了方便参考，下面列出了原来的结构：

```
struct student {
    int grade;
    int studentID;
    string name;
};
```

即使是这种形式的简单结构，仍然实现了封装。记得在第 3 章中我们用这个结构创建了一个学生数组。如果不使用这个结构，我们就不得不创建 3 个平行的数组，分别表示成绩、ID 和姓名，这显然是非常笨拙的方法。但是，我们在这个结构中并没有实现信息隐藏。基本的类框架通过把所有的数据声明为私有部分，然后通过添加公共数据的方法，允许客户代码间接引用或修改数据，从而实现了信息隐藏。

```
class studentRecord {
❶ public:
   ❷studentRecord();
    studentRecord(int newGrade, int newID, string newName);
   ❸int grade();
   ❹void setGrade(int newGrade);
    int studentID();
    void setStudentID(int newID);
    string name();
    void setName(string newName);
❺ private:
   ❻int _grade;
    int _studentID;
    string _name;
};
```

正如预想的一样，这个类的声明被分隔为一个包含成员函数的公共部分❶和一个包含了与原来的结构相同的数据❻的私有部分❺。这个类共有 8 个成员函数：2 个构造函数❷，接下来的每对成员函数分别对应于一个数据成员。例如，`_grade` 数据成员具有 2 个相关联的成员函数 `grade`❸和 `setGrade`❹。第 1 个成员函数提取一个特定的 `studentRecord` 的成绩，第 2 个成员函数为这个特定的 `studentRecord` 存储一个新的成绩。

由于与数据成员相关联的提取和存储方法极为常见，因此它们一般通过快捷术语

get 和 set 表示。正如读者所看到的那样，我把 set 这个单词并入到了向数据成员存储新值的方法名中。许多程序员还把 get 也并入到获取数据成员值的方法中，例如用 getGrade 替代 grade。我为什么没有采用这种做法呢？因为我觉得这个成员函数应该用名词而不是动词来表示。但是，有些人觉得 get 这个术语已经被使用得极为广泛并且深入人心，因此它的含义是极为清晰的，所以值得使用。总而言之，这只不过是个人风格的问题。

尽管我早就指出了 C++相对于其他语言的一些优点，但是必须承认像 C#这样的新式语言在 get 和 set 方法上有胜过 C++的独到之处。C#具有称为属性的内在机制，可以同时作为 get 和 set 方法使用。一经定义之后，客户代码可以访问属性，就像它是数据成员而不是函数调用一样。这个特性是对代码的可读性和表达能力的极大改进。在 C++中，由于缺乏这种内在机制，因此为这些方法制订命名约定以保持一致是极为重要的。

注意 我所采用的命名约定还扩展到了数据成员。和原来的结构不同，现在所有的数据成员都以下划线开头。这样就可以把 get 函数命名为与它们所提取的数据成员相同（或几乎相同）的名称。这种约定还使我们更容易识别代码中对数据成员的引用，从而提高了可读性。有些程序员使用关键字 this 进行所有数据成员的引用，而不是使用下划线前缀。因此，它不是采用下面的写法：

```
return _grade;
```

而是写成：

```
return this.grade;
```

读者此前可能还没有见过 this 关键字，它是对其所在的对象的引用。因此，如果上面这条语句出现在一个类方法中，并且这个方法还声明了一个名称为 grade 的局部变量，this.grade 这个表达式将表示数据成员 grade，而不是同名的局部变量。按照这种方式使用这个关键字在具有语法自动完成的开发环境中具有一个优点：程序员只要输入 this 并按点号键，然后从一个列表中选择数据成员，就可以避免额外的输入和潜在的拼写错误。不过，上述两种方法都强调了对数据成员的引用，这是最为重要的一点。

讨论了类的声明之后，让我们观察方法的实现。首先从第 1 对 get/set 方法开始。

```
int studentRecord::grade() {
   ❶return _grade;
}
void studentRecord::setGrade(int newGrade) {
   ❷_grade = newGrade;
}
```

这个最基本形式的 get/set 对。第 1 个方法 grade 函数返回相关联的数据成员_grade 的当前值❶。第 2 个方法 setGrade 函数把参数 newGrade 函数的值赋值给数据成员 _grade❷。但是，如果这些就是我们对类进行的所有操作，那么我们并没有实现什么。尽管这段代码提供了信息隐藏（因为它在两个方向上都支持数据的传递，不需要任何考虑或修改），但它仅仅比把_grade 声明为公共成员稍好一点而已（因为它仍然允许我们修改数据成员的名称或类型）。setGrade 方法至少应该执行一些基本的验证。它应该防止把不合理的 newGrade 值作为成绩赋值给_grade 数据成员。但是，我们必须注意遵循问题规范，不要根据自己的经验对数据做出假设，而是应该从用户的角度进行考虑。把成绩限定在 0～100 之间可能是合理的，但也可能不合理。例如，如果一所学校允许附加分，这样总分有可能会超过 100。或者学校也可能使用-1 这个成绩表示缺考。在这种情况下，由于我们通过问题描述得到了一些指导方针，因此可以把这些内容融入到数据的验证中。

```
void studentRecord::setGrade(int newGrade) {
    if ((newGrade >= 0) && (newGrade <= 100))
        _grade = newGrade;
}
```

在这里，验证仅仅是个占位的桩。但是，取决于对问题的定义，这个方法可能需要生成一条错误信息、编写一个错误日志或者对错误进行处理。

其他的 get/set 对的工作原理也是一样的。在一所特定的学校中，学生 ID 值的结构毫无疑问是存在规则的，可以据此对学生 ID 进行验证。但是，对于学生的姓名，我们除了拒绝那些带有奇特字符（例如%或@）的字符串之外就没有什么好的办法了。在当今这个充斥非主流的时代里，即使是上面这种做法似乎也是不可行的。

完成这个类的最后一个步骤是编写构造函数。在基本的类框架中，我们包含了 2 个构造函数：一个是默认构造函数，它不接受任何参数，并把数据成员设置为合理的默认值。另一个构造函数接受对应于每个数据成员的参数。第 2 个构造函数的形式对于实现表达能力这个目标是极为重要的，因为它允许我们通过一个步骤创建这个类的对象并对它的数据成员进行初始化。在编写了其他方法的代码之后，完成第 2 个构造函数就是轻

而易举的事情。

```
studentRecord::studentRecord(int newGrade, int newID, string newName) {
    setGrade(newGrade);
    setStudentID(newID);
    setName(newName);
}
```

如上所示，这个构造函数只是为每个参数调用适当的 set 方法。在大多数情况下，这是正确的方法，因为它避免了复制代码，并保证了构造函数能够利用 set 方法中的所有验证代码。

默认的构造函数有时候并不直观，这不是因为代码复杂，而是因为它不会总是存在明显的默认值。在选择数据成员的默认值时，要记住用默认构造函数所创建的对象将在什么场合使用。具体地说，就是在这种场合是否存在这个类的合法的默认对象。这可以告诉我们是用实用的默认值填充数据成员，还是用那些能够明确标明对象并没有被适当地初始化的值进行填充。例如，考虑一个表示某些值的集合的类，它封装了一个链表。合理的默认链表是存在的，也就是空链表，因此我们将设置数据成员，创建一个合法的、但在概念上为空的链表。但是，在我们的基本类示例中，并没有默认学生的合理定义。我们不想为默认的 studentRecord 对象提供一个合法的 ID 值，因为这可能与一个合法的 studentRecord 混淆。因此，我们应该为_studentID 字段选择一个明显不合法的默认值，例如-1：

```
studentRecord::studentRecord() {
    setGrade(0);
    setStudentID(-1);
    setName("");
}
```

我们用 setGrade 函数对 grade 进行赋值，这个方法会对它的参数执行验证。这意味着我们必须传递一个合法的成绩，在此例中为 0。由于 ID 被设置为一个非法的值，因此这条记录作为整体可以很容易地被确认为非法的。因此，合法的 grade 并不会产生问题。如果存在这个顾虑，也可以直接为_grade 数据成员设置一个非法的值。

这样就完成了基本的类框架。我们已经有了一组私有数据成员，表示同一个逻辑对象（在此例中为学生的班级记录）的属性。我们已经有了一些成员函数用于提取或更改对象的数据，并在适当的时候执行验证。我们还拥有了一组实用的构造函数。现在，我们已经为这个类打下了一个良好的基础。问题在于，我们是否需要进一步深入？

5.3.3 支持方法

支持方法并不仅仅是从类中提取或存储数据的方法。有些程序员可能把它们称为帮助方法、辅助方法或其他名称。但是，不管被称做什么，它们已经使类超出了基本的类框架。一组定义良好的方法常常使类变得真正实用。

为了确定可行的支持方法，需要考虑这个类将被怎样使用。我们是否期望客户代码对这个类的数据执行一些常见的操作？在这个例子中，我们已知使用这个类的程序不仅需要显示数值成绩，还需要显示字母成绩。因此，我们将创建一个支持方法，以字母的形式返回学生的成绩。首先，我们在类声明的公共部分添加对这个方法的声明。

```
string letterGrade();
```

现在，我们需要实现这个方法。该函数将根据这个问题所显示的成绩表，把存储在_grade 中的数值成绩转换为合适的字符串。我们可以用一系列的 if 语句来完成这个任务，但还有没有更清晰、更优雅的方式呢？如果读者已经想到："这看上去就像是我们在第 3 章中把收入转换为营业执照分类的方法"，那么祝贺你，你已经发现了一种恰当的编程类比。我们可以调整代码，用平行的 const 数组存储字母成绩以及与这些成绩相关联的最低数值成绩，用一个循环来转换数值成绩。

```
string studentRecord::letterGrade() {
    const int NUMBER_CATEGORIES = 11;
    const string GRADE_LETTER[] = {"F", "D", "D+", "C-", "C", "C+", "B-", "B", "B+", "A-", "A"};
    const int LOWEST_GRADE_SCORE[] = {0, 60, 67, 70, 73, 77, 80, 83, 87, 90, 93};
    int category = 0;
    while (category < NUMBER_CATEGORIES && LOWEST_GRADE_SCORE[category] <= _grade)
        category++;
    return GRADE_LETTER[category - 1];
}
```

这个方法是直接从第 3 章的函数调整而来，因此对这段代码的工作原理没什么需要解释的。但是，从函数调整为类方法还是引进了一些设计决策。首先需要注意的是我们还没有创建一个新的数据成员来存储字母成绩，而是随时为每个请求计算适当的字母成绩。另一种方法是声明一个 _letterGrade 数据成员并重写 setGrade 方法，在更新_grade 的同时更新_letterGrade。这样，letterGrade 方法将成为一个简单的 get 方法，返回已经计算产生的数据成员的值。

这种方法所存在的问题是数据冗余性。这个术语描述了数据的一种存储情况，冗余数据可能是文字数据的单纯重复，也可能是根据其他数据直接计算产生。这个问题最常见于数据库，

数据库设计者采用了详尽的过程来避免在他们的表中出现冗余数据。但是，如果我们不够小心，数据冗余还是会在任何程序中出现。为了了解这种危险，我们可以观察一个医疗记录程序，它存储了一组患者的年龄和出生日期。如果这两个值出现不一致的情况（除非年龄会自动更新，否则这种情况肯定会出现）会怎么样呢？我们该相信哪个值？冗余数据是时不时会出现的麻烦事。唯一可以为冗余数据进行辩护的理由是性能，例如对_grade 的更新极少发生而对letterGrade 函数的调用则经常进行。但是，很难想象它对于程序的整体性能会有多大的提升。

但是，这个方法可以得到改进。在测试这个方法时，我注意到了一个问题。尽管这个方法能够为合法的_grade 值产生正确的结果，但是当_grade 是负值时，这个方法就会崩溃。当 while 循环的时候，_grade 的负数会导致循环测试立即失败。因此，变量 category 保持为0，并且return语句试图引用GRADE_LETTER[-1]。我们可以通过把变量category初始化为 1 而不是 0 来避免这个问题，但这意味着负值的成绩将被设置为“F”，而它实际上不应该被设置为任何字符串，因为它是个非法的成绩值，并不对应于任何字母成绩。

反之，我们可以在把_grade 转换为字母成绩之前对它进行验证。我们已经在 setGrade 方法中对成绩值进行了验证。因此，我们并不需要向 letterGrade 方法添加新的验证代码，而是应该“重构”出这两个方法中的公共代码，用于创建第 3 个方法。（读者可能会疑惑，如果我们在设置成绩时对它进行验证，可能会遇到非法的成绩。但是，记住默认构造函数用-1表示还没有设置合法的成绩值。）这是另一种类型的支持方法，相当于前面章节所介绍的基本帮助函数的类版本。让我们实现这个方法，并修改其他方法以配合使用这个方法：

```
❶ bool studentRecord::❷isValidGrade(❸int grade) {
    if ((grade >= 0) && (grade <= 100))
        return true;
    else
        return false;
}
void studentRecord::setGrade(int newGrade) {
    if (❹isValidGrade(newGrade))
        _grade = newGrade;
}
string studentRecord::letterGrade() {
    if (❺!isValidGrade(_grade)) return "ERROR";
    const int NUMBER_CATEGORIES = 11;
    const string GRADE_LETTER[] = {"F", "D", "D+", "C-", "C", "C+", "B-", "B", "B+", "A-", "A"};
    const int LOWEST_GRADE_SCORE[] = {0, 60, 67, 70, 73, 77, 80, 83, 87, 90, 93};
    int category = 0;
    while (category < NUMBER_CATEGORIES && LOWEST_GRADE_SCORE[category] <= _grade)
        category++;
    return GRADE_LETTER[category - 1];
}
```

这个新的验证方法的类型是 bool❶，由于这是个“是”或“否”的问题，所以我为它取名为 isValidGrade❷。这个名称与日常的英语表示方法非常接近，很容易想到在 setGrade❹和 letterGrade❺方法中调用它。另外，这个方法接受需要验证的成绩作为它的参数❸。尽管 letterGrade 方法所验证的是_grade 数据成员已经存储的值，但 setGrade 方法所验证的值可能会赋值给这个数据成员，也可能不赋值给它。因此，isValidGrade 方法需要接受成绩作为参数，使之可以被其他两个方法所使用。

尽管已经实现了 isValidGrade 方法，但还是有一个与它相关的决定尚未做出：它应该被设置为什么访问级别？也就是说，我们应该把它放在类的公共部分还是私有部分？和基本类框架中总是出现在公共部分的 get 和 set 方法不同，支持方法可以出现在公共部分，也可以出现在私有部分，具体取决于它们的用途。

把 isValidGrade 放在公共部分会有什么效果呢？最显而易见的是，客户代码可以访问这个方法。由于更多的公共方法使类更为实用，因此许多程序员新手把每个方法都放在公共部分，使它们可以被类的客户所调用。但是，这就忽略了公共访问设置的另一个效果。记住，公共部分定义了类的接口，当类集成到一个或多个程序中时，应该尽量避免对这些方法进行修改，因为这种修改很可能产生级联效应，导致所有的客户代码都需要修改。

因此，把方法放在公共部分就锁定了这个方法的接口和它的效果。在这个例子中，假设有些客户代码根据 isValidGrade 方法的最初格式，认定它是对 0～100 之间的成绩进行验证，但是后来可接受的成绩的范围变得更为复杂，这样客户代码可能会失败。为了避免这个问题，我们可能不得不在类中创建第 2 个成绩验证方法，而把第 1 个验证方法晾在一边。

假设我们期望限制客户使用 isValidGrade 方法，而决定不把它放在公共部分。我们可以把这个方法声明为私有部分，但这并不是唯一的选择。由于此函数并没有直接引用这个类的任何数据成员或其他任何方法，因此我们可以在这个类的外部声明这个函数。但是，这种做法不仅产生了公共访问所存在的修改性问题，还降低了代码的封装性，因为这个函数现在是这个类所需要的，却不再是它的组成部分。我们还可以让这个方法留在类中，但是把它声明为保护成员而不是私有成员。

保护成员和私有成员的区别是在子类中体现的。如果 isValidGrade 是个保护方法，这个方法可以被子类中的方法所调用。如果 isValidGrade 是个私有方法，它只能被 studentRecord 类的其他方法所调用。这相当于在更小规模上公共部分与私有部分的区

别。我们是期望子类中的方法更大限度地利用自己的方法，还是期望这个方法的效果或它的接口在未来可能发生变化？在许多情况下，最安全的做法是把所有的帮助函数声明为私有部分，只把那些专门为了方便客户而编写的支持函数声明为公共部分。

5.4 具有动态数据的类

创建类的最好理由之一就是封装动态数据结构。正如我们在第 4 章所讨论的那样，程序员在现实中面临追踪动态内存分配、指针赋值和销毁等困难繁琐的工作，以避免内存泄漏、野引用和非法的内存引用。把所有的指针引用放在一个类中并不能消减工作的难度，但它意味着我们能够让其保持正确，并且可以安全地把相关的代码放到其他项目中。它还意味着与动态数据结构相关的所有问题都被隔离到类内部本身的代码中，从而简化了代码的调试。

让我们创建一个具有动态数据的类，观察它的工作原理。对于这个示例问题，我们打算使用第 4 章的主要问题的一个修改版本。

追踪未知数量的学生记录

在这个问题中，我们将编写一个类，它包含了用于存储和操纵一个学生记录集合的方法。一条学生记录包含了一个学生编号和一个成绩，它们都用整数表示，另外还有一个字符串表示学生的姓名。我们需要实现下面这些函数：

addRecord：这个函数接受一个学生编号、姓名和成绩为参数，并用这些信息向集合中添加一条新记录。

recordWithNumber：这个函数接受一个学生编号为参数，并从集合中提取指定学生编号的那条记录。

removeRecord：这个函数接受一个学生编号为参数，并从集合中删除指定学生编号的那条记录。

这个集合可以是任意长度。addRecord 操作预计会被频繁调用，因此它必须高效地实现。

这段描述与最初版本的主要区别是我们添加了一个新操作 record WithNumber，另

外所有的操作都不需要引用任何指针参数。这是用类封装链表的关键优点。客户可能会意识到这个类以链表的形式实现了学生记录集合，甚至可能依赖于这种实现（记住我们前面关于信息隐藏的限制的讨论）。但是，客户代码并不会与这个类中的链表或任何指针直接交互。

由于这个问题为每个学生所存储的信息的内容与之前的问题的内容是相同的，因此我们有机会在这里实现类的复用。在我们的链表节点类型中，并不是用单独的字段表示 3 段学生数据中的每一段，而是使用了一个 studentRecord 对象。一个类把另一个类的对象作为它的数据成员的方式称为合成。

现在，我们已经拥有了足够的信息，可以实现初步的类声明：

```
class studentCollection {
private:
  ❶struct studentNode {
      ❷studentRecord studentData;
        studentNode * next;
    };
❸ public:
      studentCollection();
      void addRecord(studentRecord newStudent);
      studentRecord recordWithNumber(int idNum);
      void removeRecord(int idNum);
  private:
    ❹typedef studentNode * studentList;
    ❺studentList _listHead;
  };
```

我以前曾经说过，程序员倾向于让类的公共声明出现在前面，但这里我们做了特殊处理。我们首先进行了节点结构 studentNode 的私有声明❶，我们将用它来创建链表。这个声明必须出现在公共部分之前，因为有几个公共成员函数需要引用这个类型。和第 4 章所讨论的节点类型不同，这种节点并没有使用单独的字段表示负载数据，而是包含了一个 studentRecord 结构类型的成员❷，公共成员函数❸直接出现在问题描述之后。另外，和往常一样，我们还声明了一个构造函数。在第 2 个私有部分中，我们声明了一个 typedef❹，表示指向节点类型的指针。这是为了清晰起见，就像我们在第 4 章中的做法一样。接着，我们声明了链表的头指针，非常合理地把它命名为_listHead❺。

这个类声明了两个私有类型。除了声明成员函数和数据成员之外，类也可以声明类型。和其他成员一样，出现在类中的类型可以用任何访问指示符声明。但是，和数据成员一样，在默认情况下应该考虑把类型定义声明为私有，只有在拥有十分充足的理由时才能放松这

个限制。类型声明一般是类在幕后进行操作的核心，因此它们对于信息隐藏是十分关键的。而且，在大多数情况下，客户代码并不会使用在类中声明的类型，唯一的例外是类中所定义的一个类型作为一个公共方法的返回类型或者作为一个公共方法的参数类型。在这种情况下，这个类型必须被声明为公共部分，否则这个公共方法将不能被客户代码所使用。studentCollection 类假设结构类型 studentRecord 是被独立声明的，但我们也可以把它作为这个类的一部分。如果采用了这种做法，就必须在类的公共部分声明它。

现在我们可以实现这个类的方法了，首先从构造函数开始。与此前的例子不同，在这里我们只声明了一个默认构造函数，而不是接受参数并对数据成员进行初始化的构造函数。这个类的总体目标是隐藏链表的细节，因此我们甚至不想让客户意识到_listHead 的存在，更不用说让客户对它进行操作了。我们在这个默认构造函数中需要执行的任务就是把头指针设置为 NULL。

```
studentCollection::studentCollection() {
    _listHead = NULL;
}
```

添加节点

现在我们把目光转移到 addRecord 函数。由于这个问题的描述并没有要求按任何特定的顺序存放学生记录，因此我们直接采用了第 4 章的 addRecord 函数。

```
void studentCollection::addRecord(❶studentRecord newStudent) {
    ❷studentNode * newNode = new studentNode;
    ❸newNode->studentData = newStudent;
    ❹newNode->next = _listHead;
    ❺_listHead = newNode;
}
```

这段代码和我们的蓝图函数只有两处不同。现在，我们在参数列表中只需要一个参数❶，它是需要添加到集合中的 studentRecord 对象。这个对象封装了一位学生的所有数据，从而减少了所需要的参数数量。另外，我们并不需要向这个函数传递一个链表头指针，因此它已经作为_listHead 存储在这个类中，可以根据需要直接引用。和第 4 章的 addRecord 函数一样，我们创建了一个新节点❷，把新的学生数据复制到这个新节点中❸，使新节点的 next 字段指向链表中以前的第 1 个节点❹，最后把_listHead 指向这个新节点❺。一般情况下我推荐为所有的指针操作绘制一张图，但由于它与以前所执行的操作相同，因此可以参考以前所绘制的图。

现在我们可以把注意力转向 3 个成员函数的最后一个，即 record WithNumber。这

个名称稍稍有点绕口，有些程序员可能会选择 retrieveRecord 或类似的名称。但是，根据以前所陈述的命名规则，我还是决定使用名词，因为这个方法会返回一个值。这个方法与 averageRecord 方法的相似之处在于它也需要遍历整个链表。对于这个函数而言，区别在于一旦找到了匹配的学生记录之后就停止遍历。

```
studentRecord studentCollection::recordWithNumber(int idNum) {
  ❶studentNode * loopPtr = _listHead;
  ❷while (loopPtr->studentData.studentID() != idNum) {
       loopPtr = loopPtr->next;
    }
  ❸return loopPtr->studentData;
}
```

在这个函数中，我们把循环指针初始化为链表的头指针❶，并对链表进行遍历，直到找到目标 ID 号❷为止。最后，到达目标节点时，返回整条匹配的记录作为这个函数的返回值❸。这段代码看上去很不错，但是和往常一样，我们必须考虑潜在的特殊情况。在处理链表中总是需要考虑的情况是在头指针一开始为 NULL 的情况。对于这个函数，面临这种情况时肯定会产生问题，因为我们并没有检查这种情况。当我们在进入循环时首先对指针 loopPtr 进行解引用，代码就会崩溃。但是，从更广泛的意义上说，我们必须考虑客户代码所提供的 ID 号不一定与集合中的任何记录匹配的可能性。在这种情况下，即使_listHead 并不是 NULL，当我们到达链表的尾部时，指针 loopPtr 最终也将会变成 NULL。

因此，基本的问题是我们需要在指针 loopPtr 变成 NULL 时终止循环。这并不困难，但是在这种情况下我们该返回什么呢？显然我们无法返回 loopPtr->studentData，因为 loopPtr 将是 NULL。反之，我们可以创建并返回一个包含明显非法值的哑 studentRecord 对象。

```
studentRecord studentCollection::recordWithNumber(int idNum) {
    studentNode * loopPtr = _listHead;
    while (❶loopPtr != NULL && loopPtr->studentData.studentID() != idNum) {
        loopPtr = loopPtr->next;
    }
    if (❷loopPtr == NULL) {
      ❸studentRecord dummyRecord(-1, -1, "");
        return dummyRecord;
    } else {
        return loopPtr->studentData;
    }
}
```

在这个版本的方法中，如果当循环结束时❷指针为 NULL，我们就创建一条哑记录，用空字符串表示姓名，用-1 表示成绩和学生 ID 的值❸，并返回这条哑记录。回到循环内部，我们检查指针 loopPtr 为 NULL 的情况，这可能是由于不存在需要遍历的链表，或者由于遍历完了链表之后没有成功地找到匹配的记录。这里需要注意的一个关键在于循环的条件表达式❶是个复合表达式，首先出现的是 loopPtr != NULL，这是极为重要的。对复合布尔型表达式进行求值时，C++使用了一种称为短路求值的机制。简单地说，当整个表达式的值已知时，它就不会对复合布尔表达式的右半部分进行求值。例如，&&表示逻辑与运算符，如果一个&&表达式的左边的求值为 false，不管表右边求值的结果是什么，整个表达式的结果肯定是 false。为了追求效率，C++利用了这个事实，当&&表达式的左侧为 false 时就跳过右侧的求值（基于相同的原因，对于逻辑或运算符||，当左侧的结果确实为 true 时，右侧也不再进行求值）。因此，当 loopPtr 为 NULL 时，表达式 loopPtr != NULL 的结果为 false，这样&&的右边就不会被求值。如果没有短路求值机制，表达式的右边也会被求值，这将会导致对 NULL 指针的解引用，从而导致程序崩溃。

这种实现虽然避免了第 1 个版本的潜在崩溃危险，但我们还是需要注意到它对客户代码寄予了极大的信任。也就是说，调用这个方法的函数要负责检查返回的 studentRecord 对象，确保它不是哑记录才能对其进一步处理。如果读者也像我这样悲观，可能会对这种假设感觉不安。

异常

这是另一种选项。和其他许多编程语言一样，C++提供了一种称为异常的机制，允许函数（方法或通用函数）无歧义地向调用者表示一种错误状态。它适用于与这个方法相似的类型的场合，也就是在接收到不良输入时没有合适的东西可以返回。我们不准备在这里讨论异常的语法，并且令人遗憾的是，按照 C++对异常的实现方式，它们并不能真正解决前面这个段落所描述的真正问题。

重新排列链表

removeRecord 方法与 recordWithNumber 方法的相似之处在于我们必须遍历链表才能找到需要从链表中删除的节点，但除此之外还有很多值得注意的地方。从链表删除一个节点要求精心维护链表中的剩余节点。填补这种方法所创建的空隙的最简单方法，就是把被删除节点之前的那个节点链接到它之后的那个节点。我们并不需要设计一个函数，因

为我们已经在类声明中有了这样一个函数，因此只需要一个测试用例：

```
studentCollection s;
studentRecord stu3(84, 1152, "Sue");
studentRecord stu2(75, 4875, "Ed");
studentRecord stu1(98, 2938, "Todd");
s.addRecord(stu3);
s.addRecord(stu2);
s.addRecord(stu1);
❶ s.removeRecord(4875);
```

我们在这里创建了一个 studentCollection 对象 s 以及 3 个 student Record 对象，每个对象将被添加到这个集合中。注意，我们可以复用同一条记录，在几次调用 addRecord 时改用不同的值，但是上面这种做法对于我们的测试代码而言显然更加简单。测试代码的最后一行调用了 removeRecord❶函数，它将在此时删除第 2 条记录，也就是学生姓名为“Ed.”的那条记录。按照与第 4 章所描述的指针图相同的风格，图 5.1 显示了这次调用之前和之后的内存状态。

在图 5.1（a）中，我们看到了测试代码所创建的链表。注意，由于我们使用了一个类，因此这张图略微偏离了原先的约定。在堆栈/堆分支的左边是 _listHead，它是 studentCollection 对象 s 内部的私有数据成员，另外还有 idNum，它是 removeRecord 方法的参数。右边是链表本身，它位于堆中。记住，addRecord 方法把新记录放在链表的起始处，因此链表的出现顺序与测试代码中它们添加到链表的顺序是相反的。中间那个节点“Ed”的 ID 号与参数 4875 匹配，因此将它从链表中删除。图 5.1（b）显示了这个调用的结果。链表中的第 1 个节点“Todd”现在指向列表的第 3 个节点“Sue”。“Ed”节点不再链接到这个链表中，而是已经被删除。

既然我们已经知道了代码应该产生什么效果，现在就可以编写它了。由于我们知道了需要寻找具有匹配 ID 号的节点，因此可以从 recordWithNumber 函数的 while 循环开始。当这个循环结束时，我们就得到了一个指向需要删除的节点的指针。遗憾的是，为了完成删除任务，光有这个指针还是不够的。如图 5.1 所示，为了填补空隙并维护这个链表，我们需要修改“Todd”节点的 next 字段。如果我们只有一个指向“Ed”节点的指针，就没有办法引用“Todd”节点，由于链表中的每个节点只能引用它的后继节点，而不能引用它的前驱节点。（为了处理类似这样的情况，有些链表实现了双向链接。这类链表称为双链表，但它们极少被使用。）因此，除了指向需要被删除的节点的指针（如果我们沿用前一个函数的代码，它将被称为 loopPtr）之外，我们还需要一个指向它之前的那个节点的指针。我们把这个指针称为 trailing。图 5.2 显示了把这个概念应用于这个测试用

例的结果。

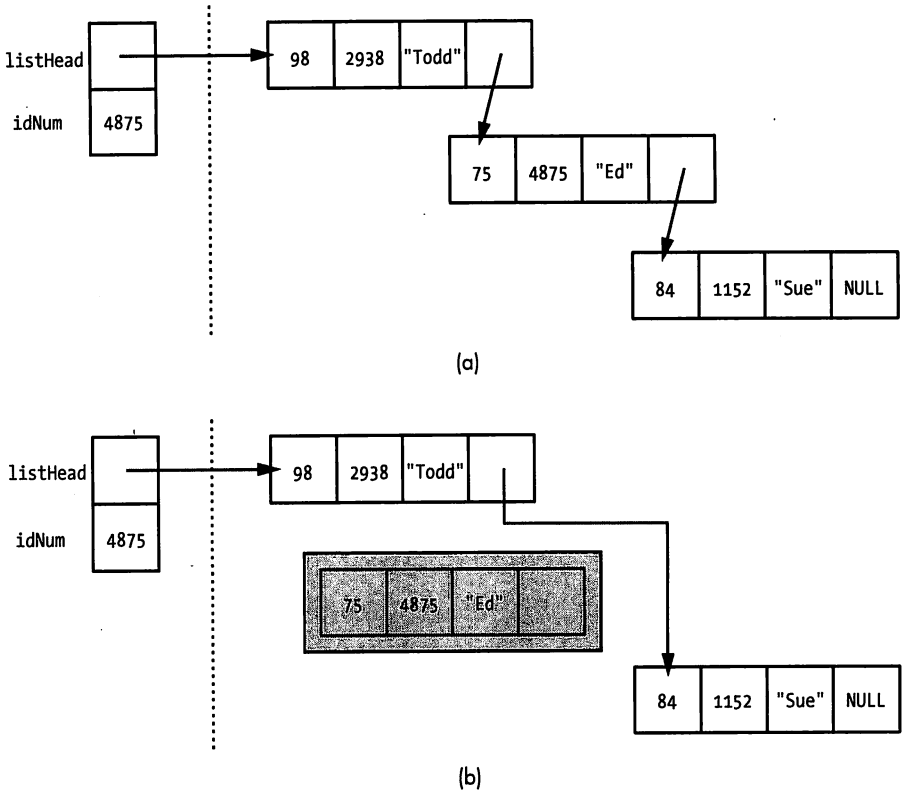

图 5.1 removeRecord 测试例的“之前”（a）和“之后”（b）状态

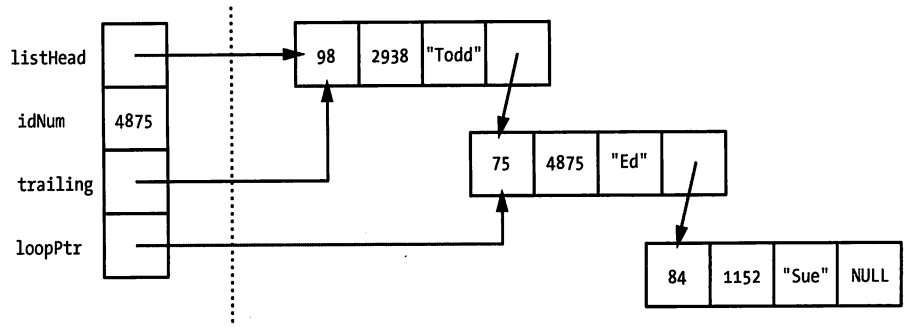

图 5.2 删除由 idNum 所指向的节点所需要的指针

有了 `loopPtr` 引用需要被删除的节点以及 `trailing` 引用之前的那个节点之后，我们就可以删除所需的节点并维持链表的完整性了。

```
void studentCollection::removeRecord(int idNum) {
    studentNode * loopPtr = _listHead;
  ❶studentNode * trailing = NULL;
    while (loopPtr != NULL && loopPtr->studentData.studentID() != idNum) {
       ❷trailing = loopPtr;
        loopPtr = loopPtr->next;
    }
  ❸if (loopPtr == NULL) return;
  ❹trailing->next = loopPtr->next;
  ❺delete loopPtr;
}
```

这个函数的第 1 部分与 recordWithNumber 函数相似，区别在于我们声明了 trailing 指针❶，并且在这个循环的内部把指针 loopPtr 的旧值赋值给 trailing❷，然后再让 loopPtr 指向下一个节点。按照这种方式，trailing 始终指向 loopPtr 之前的那个节点。由于我们是在前一个函数的基础之上进行操作的，所以已经处理了一种特殊情况。因此，当循环结束时，我们检查 loopPtr 的值是否为 NULL。如果是，意味着没有找到与目标 ID 号匹配的节点，函数将立即返回❸。我把出现在函数中间的 return 语句称为“提前逃避”。有些程序员反对这种做法，因为具有多个退出点的函数比较难以让人理解。但是对这个例子而言，如果采用替代方案，则要求在后面的 if 语句中添加另一层嵌套，我个人认为采用“提前逃避”的方法更好一点。

确定了存在需要删除的节点之后，就可以删除它了。根据我们的图，可以看到需要把 trailing 所指向节点的 next 字段设置为 loopPtr 节点的 next 字段当前所指向的节点❹。然后，我们就可以安全地删除 loopPtr 所指向的节点了❺。

这个函数对于我们的测试用例而言没有问题。但是和往常一样，我们需要检查潜在的特殊情况。我们已经处理了 idNum 变量值与集合中的任何记录都不匹配的可能性，但是不是还有其他可能出现的问题呢？仔细观察这个测试用例，如果我们删除第 1 个节点或最后一个节点而不是中间那个节点，事情会发生什么变化吗？通过测试和手工检查，我们发现删除第 3 个节点（即最后那个节点）不会产生问题。但是，删除第 1 个节点确实产生了一点麻烦，因为在这种情况下，不存在可以让 trailing 所指向的前一个节点。因此，我们必须对_listHead 本身进行操作。图 5.3 显示了 while 循环结束时的情况。

在这种情况下，我们需要把_listHead 重新指向列表中以前的第 2 个节点，也就是“Ed”节点。我们重新编写这个方法，以处理这种特殊情况。

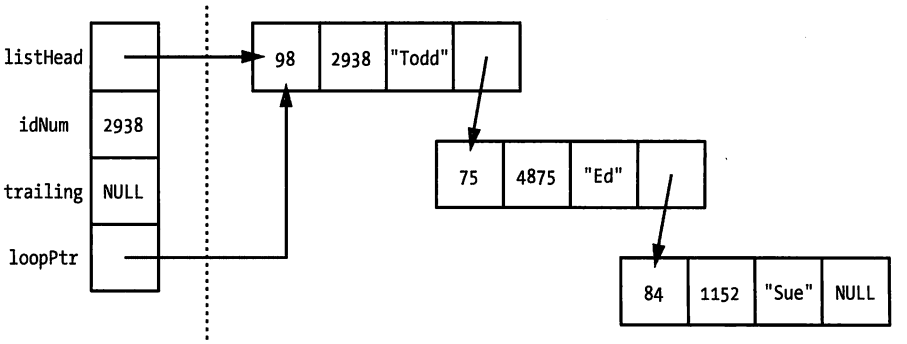

图 5.3 删除链表的第 1 个节点之前的情况

```
void studentCollection::removeRecord(int idNum) {
    studentNode * loopPtr = _listHead;
    studentNode * trailing = NULL;
    while (loopPtr != NULL && loopPtr->studentData.studentID() != idNum) {

        trailing = loopPtr;
        loopPtr = loopPtr->next;
    }
    if (loopPtr == NULL) return;
  ❶if (trailing == NULL) {
      ❷_listHead = _listHead->next;
    } else {
        trailing->next = loopPtr->next;
    }
    delete loopPtr;
}
```

正如我们所看到的那样，条件测试❶和处理特殊情况的代码❷都非常简明，因为我们在编写代码之前已经对情况进行了精心的分析。

析构函数

实现了这个问题所指定的 3 种方法之后，我们可能觉得 `student Collection` 类已经完成了。但是，它还存在严重的问题。第一个问题是这个类还缺少一个析构函数。这是一种特殊的方法，是当对象离开作用域（当声明这个对象的函数结束执行时）时被调用的。当一个类没有动态数据时，一般并不需要析构函数。但是如果类中具有动态数据，就需要定义析构函数。记住，我们必须删除用 new 所分配的所有内存，以避免内存泄漏。如果 `studentCollectionod` 类的一个对象具有 3 个节点，则每个节点都需要被销毁。但是，我们并不需要直接执行这个任务，而是通过编写一个帮助方法删除一个 `studentList` 中的所有节点。在这个类的私有部分，我们添加了下面的声明：

```
void deleteList(studentList &listPtr);
```

这个方法本身的代码如下：

```
void studentCollection::deleteList(studentList &listPtr) {
    while (listPtr != NULL) {
        ❶studentNode * temp = listPtr;
        ❷listPtr = listPtr->next;
        ❸delete temp;
    }
}
```

对链表进行遍历时，把指向当前节点的指针复制到一个临时变量❶，把当前节点指针向后推进一个节点❷，然后删除这个临时变量所指向的节点❸。有了这段代码之后，我们可以非常简单地编写析构函数。首先，我们在类声明的公共部分添加这个析构函数：

```
~studentCollection();
```

注意 和构造函数相似，析构函数也是用类名表示的，但它没有返回类型。名称前面的波浪号把析构函数和构造函数区分开来。它的实现如下所示：

```
studentCollection::~studentCollection() {
    deleteList(_listHead);
}
```

虽然这些方法中的代码非常简单，但是对析构函数进行测试是非常重要的。尽管编写不佳的析构函数可能会导致程序崩溃，但许多析构函数问题并不会引起程序崩溃，只是导致内存泄漏或者出现更糟的情况，甚至出现无法解释的程序行为。因此，利用自己的开发环境的调试器对析构函数进行测试是非常重要的，确保看到析构函数确实对每个节点调用了 `delete` 操作。

深拷贝

现在还有一个严重的问题。在第 4 章中，我们简单讨论了交叉链接的概念，即两个指针变量具有相同的值。即使这些指针变量本身并不相同，但它们指向同一个数据结构。因此，修改一个指针所指向的结构会导致另一个指针也被修改。在进行了动态内存分配的类中很容易出现这个问题。为了观察这个问题，考虑下面这些基本的 C++代码序列：

```
int x = 10;
int y = 15;
x = y;
❶ x = 5;
```

如果我提问最后一条语句对变量 y 的值会产生什么影响时，读者很可能会觉得我是不是想多了。最后一条语句不会对 y 产生任何影响，只会对 x 产生影响。但是，现在考虑如下所示的代码：

```
studentCollection s1;
studentCollection s2;
studentRecord r1(85, 99837, "John");
s2.addRecord(r1);
studentRecord r2(77, 4765, "Elsie");
s2.addRecord(r2);
❶ s1 = s2;
❷ s2.removeRecord(99837);
```

假设我所提的问题是最后一条语句❷对 s1 有什么影响。那么，它确实会产生影响。尽管 s1 和 s2 是两个不同的对象，但它们并不是完全独立的对象。在默认情况下，当一个对象被复制给另一个对象时，就像把 s2 赋值给 s1 时❶，C++会执行一种称为浅拷贝的操作。在浅拷贝中，一个对象的每个数据成员被直接复制给另一个对象。因此，如果我们的唯一数据成员_listHead 是公共的，那么 s1= s2 的效果相当于 s1._listHead = s2._listHead。这使得两个对象的_listHead 数据成员指向内存中的同一个地方，也就是表示“Elsie”的那个节点，而这个节点又指向表示“John”的那个节点。因此，当“John”节点被删除时，很显然它是从两个链表中被删除的，因为两个链表实际上是同一个链表。图 5.4 显示了代码结束时的情况。

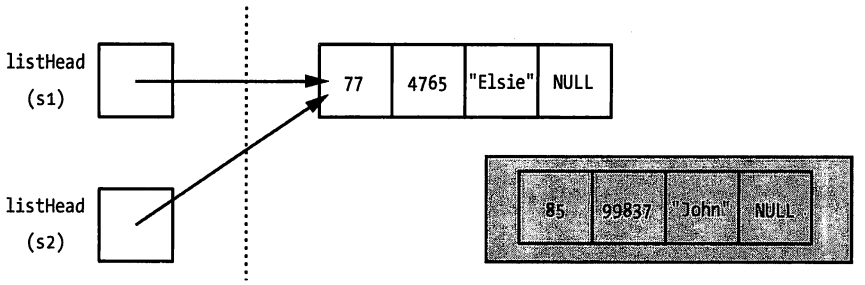

图 5.4 浅拷贝产生交叉链表，删除一个链表的“John”节点会导致两个链表的这个节点都被删除

但是，实际情况可能更糟。如果这段代码的最后一行删除了第 1 条记录即“Elsie”节点会怎样？在这种情况下，s2 中的_listHead 将因被更新而指向“John”，并且“Elsie”

节点将被删除。但是，s1 内部的_listHead 节点将仍然指向被删除的“Elsie”节点，从而成为危险的野引用，如图 5.5 所示。

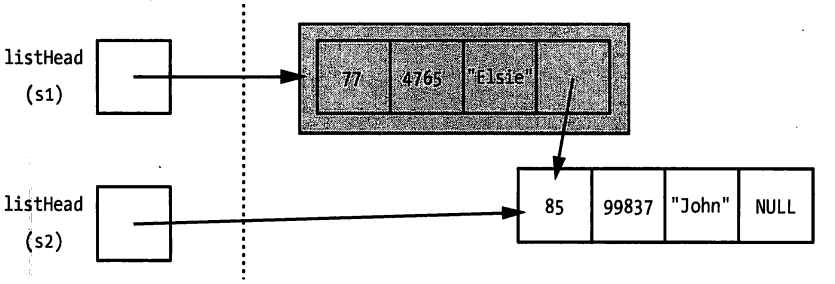

图 5.5 在 s2 中进行删除导致 s1 中出现了一个野引用

这个问题的解决方案是采用深拷贝，这意味着我们不仅要复制指向这个结构的指针，而且要为这个结构中的所有东西创建一份拷贝。在此例中，这意味着复制链表中的所有节点，创建一份真正的链表拷贝。和此前一样，我们首先创建一个私有的帮助函数，用于复制 studentList 对象。它在这个类的私有部分的声明如下：

```
studentList copiedList(const studentList original);
```

和以前一样，我选择了一个名词作为具有返回值的方法的名称。这个方法的实现如下所示：

```
❶ studentCollection::studentList studentCollection::copiedList(const studentList original) {
    ❷if (original == NULL) {
         return NULL;
      }
      studentList newList = new studentNode;
    ❸newList->studentData = original->studentData;
    ❹studentNode * oldLoopPtr = original->next;
    ❺studentNode * newLoopPtr = newList;
      while (oldLoopPtr != NULL) {
        ❻newLoopPtr->next = new studentNode;
         newLoopPtr = newLoopPtr->next;
         newLoopPtr->studentData = oldLoopPtr->studentData;
         oldLoopPtr = oldLoopPtr->next;
      }
    ❼newLoopPtr->next = NULL;
    ❽return newList;
  }
```

这个方法中包含了很多内容，下面我们逐步对它进行解释。从语法上说，在方法的实现中指定返回类型时，必须加上类名作为前缀❶。否则，编译器就不知道我们所讨论的是

什么类型。(在方法内部就没有这个必要，因为编译器已经知道这个方法属于哪个类，听上去稍稍令人困惑!）我们检查参数所表示的链表是否为空。如果为空就退出这个方法❷。如果它不为空，可以创建一份拷贝，在循环之前复制第 1 个节点的数据❸，因为我们必须根据这个节点修改新链表的头指针。

接着，我们设置 2 个指针，对两个链表进行追踪。指针 oldLoopPtr❹对作为参数所传递的列表进行遍历，它总是指向我们将要进行复制的那个节点。指针 newLoopPtr❺对通过复制产生的新链表进行遍历，它总是指向我们所创建的最后一个节点，也就是我们将要添加下一个节点的位置之前的那个节点。就像 removeRecord 方法一样，我们需要某种类型的缀尾指针。在这个循环的内部❻，我们创建了一个新节点，并使 newLoopPtr 指向它，把旧节点的数据复制到这个新节点，并把 oldLoopPtr 推进一个节点。在循环之后，我们通过把最后一个节点的 next 字段设置为 NULL 结束这个新链表❼，并返回指向这个新链表的指针❽。

那么，这个帮助函数是怎样解决我们之前所说的问题的呢？它自身并不能做到这一点，但是有了这些代码之后，我们现在就可以对赋值操作符进行重载了。操作符重载是 C++所提供的一个特性，允许我们更改内置操作符对某种类型所执行的操作。在这个例子中，我们想要重载赋值操作符（=），这样它就不会执行默认的浅拷贝，而是调用 copiedList 方法执行深拷贝。在这个类的公共部分，我们添加了下面这些代码：

```
❶ studentCollection& ❷operator=(❸const studentCollection & ❹rhs);
```

在指定操作符重载时，我们用关键字 operator 和需要重载的操作符对这个方法进行命名❷。我为参数所选择的名称（rhs❹）是操作符重载的一个常见选择，它表示右侧(right-hand side)，可以使程序员一目了然。因此，在开启这个话题的赋值语句 s2= s1 中，s1 对象将出现在赋值操作符的右边，s2 将出现在左边。我们通过参数引用右边的对象，并通过直接访问类成员引用左边的对象，这也是引用类的其他方法中的一种方式。因此，我们在这个例子中的任务就是创建由_listHead 所指向的一个链表，使它成为由 rhs 的 _listHead 所指向的链表的一份拷贝。这样，调用 s2= s1 的效果相当于使 s2 成为 s1 的一份真正拷贝。

参数的类型总是类的一个常量引用❸，返回类型也是类的一个引用❶。我们很快将会看到参数为什么是引用的原因。读者可能会奇怪这个方法为什么要有返回值，因为我们在这个方法中是对数据成员直接进行操作的。这是由于 C++允许串联式的赋值，例如 s3 = s2 = s1，这样赋值操作符的返回值可以作为另一个赋值操作符的参数。

理解了所有的语法之后，赋值操作符的代码就相当直接了：

```
studentCollection& studentCollection::operator=(const studentCollection &rhs) {
  ❶if (this != &rhs) {
      ❷deleteList(_listHead);
      ❸_listHead = copiedList(rhs._listHead);
    }
  ❹return *this;
}
```

为了避免内存泄漏，我们首先必须删除左边链表的所有节点❷。（出于此目的，我们把 deleteList 编写为帮助函数而不是把它的代码直接包含在析构函数中。）左边的链表被删除之后，我们就使用另一个帮助方法复制右边的链表❸。但是，在执行这些步骤之前，我们通过检查指向左边对象和右边对象的指针是否不同来判断这两个对象是否不同（也就是说，确保不会出现 s1 = s1 这样的情况出现）❶。如果两个指针相同，就不需要执行任何操作。但是，并不仅仅是为了效率才进行这样的检查。如果我们对同一个指针执行了深拷贝，当我们删除了左边链表当前所拥有的节点时，还将删除右边链表中的节点。最后，返回一个指向左边对象的指针❹，而不管我们实际是否复制了链表。这是因为虽然像 s2 = s1= s1 这样的语句看上去非常怪异，但是我们仍然想要确保即使真有人采用这样的写法，这个方法也能够适应。

完成了链表拷贝的帮助函数之后，我们还应该创建一个拷贝构造函数。这种构造函数接受同一个类的另一个对象作为参数。当我们需要创建一个现有的 studentCollection 对象的一份拷贝时，可以显式地调用拷贝构造函数。但是，当一个类的对象作为值参数传递给一个函数时，也会隐式地调用拷贝构造函数。因此，我们应该考虑以常量引用的形式传递对象参数，而不是值参数，除非接受这个对象的函数需要对这份拷贝进行修改。否则，代码就会执行大量不必要的操作。例如，考虑一个包含 10000 条记录的学生集合。这个集合可能通过一个指针以引用的形式传递给函数。否则，就会调用拷贝构造函数进行漫长的遍历，进行 10000 次内存分配，并且这份局部拷贝在函数结束时可能调用析构函数，再经历一遍漫长的遍历和 10000 次内存销毁。这就是为什么赋值操作符重载时右边的参数采用常量引用参数形式的原因。

为了在类中添加拷贝构造函数，首先在类声明的公共部分添加对它的声明：

```
studentCollection(const studentCollection &original);
```

和所有的构造函数一样，它也没有返回类型。另外，和重载的赋值操作符一样，它的

参数也是这个类的一个常量引用。

这个拷贝构造函数的实现相当简单，因为我们已经在帮助方法中实现了它的逻辑。

```
studentCollection::studentCollection(const studentCollection &original) {
    _listHead = copiedList(original._listHead);
}
```

现在我们可以创建一个如下的声明：

```
studentCollection s2(s1);
```

这条语句的效果是声明了 s2 并把 s1 的节点复制给它。

分配动态内存的类的概括

完成了问题描述所指定的方法之后，我们实际上已经完成了这个类的大量工作。因此，让我们花费时间对它进行回顾。下面是这个类的声明：

```
class studentCollection {
private:
    struct studentNode {
        studentRecord studentData;
        studentNode * next;
    };
public:
    studentCollection();
    ~studentCollection();
    studentCollection(const studentCollection &original);
    studentCollection& operator=(const studentCollection &rhs);
    void addRecord(studentRecord newStudent);
    studentRecord recordWithNumber(int idNum);
    void removeRecord(int idNum);
private:
    typedef studentNode * studentList;
    studentList _listHead;
    void deleteList(studentList &listPtr);
    studentList copiedList(const studentList original);
};
```

我们在这里所学到的经验是在创建一个具有动态内存的类时，需要创建一些新的片段。除了基本的类框架（私有数据、默认构造函数和用于发送对象数据的方法）之外，还需要添加一些额外的方法，处理动态内存的分配和清理。至少，我们应该添加一个拷贝构造函数和一个析构函数。如果用户可能使用赋值操作符，还需要对它进行重载。这些额外的方法常常可以通过创建用于复制或删除底层动态数据结构的帮助方法来实现。

这看上去像是要完成大量的工作，事实也可能如此。但是，要意识到我们向这个类所添加的任务本身就是自己需要处理的。换句话说，即使我们不为学生记录的链表集合创建一个类，也仍然需要负责在完成了对链表中的节点的使用之后将它们删除。我们仍然必须警惕交叉链接。如果我们想创建原链表的一份真正拷贝，仍然需要遍历这个链表并逐个复制节点。把所有的东西都放在一个类结构中所增加的工作量实在很有限。当一切完工之后，客户代码就可以忽略所有的内存分配细节。最后，封装和信息隐藏使动态数据结构变得更容易操作。

5.5 需要避免的错误

我们已经讨论了怎样用 C++创建良好的类，因此现在通过考虑一些应该避免的陷阱来完成对这个话题的讨论。

5.5.1 假类

正如我在本章之初所说的，C++是一种混合语言，它同时包含了过程性编程和面向对象编程的惯用法。它是一种非常出色的学习面向对象编程的语言，因为类的创建总是程序员角色所做出的积极选择。在类似 Java 这样的语言中，问题从来不是“我应该创建一个类吗？”而是“我怎么才能把这些东西放在类中？”把所有的东西都放在类中这个需求就导致了我所称的假类的产生，也就是语法完全正确，但是缺乏连贯的设计，使类没有真正的含义。当类这个词在编程中使用时，它表示一组具有一些共同属性的东西，设计良好的 C++类就符合这个定义。

假类可能由于几个原因而引发。一种类型的出现是因为程序员实际上想使用全局变量，这除了偷懒不想在函数之间传递参数之外，没有其他可辩解的理由（这种原因就算有也是极为罕见的）。程序员知道广泛使用全局变量是糟糕的风格，他们觉得这个漏洞很容易被发现。于是，他们把程序中的全部或大部分函数塞入到一个类中，那些原来是变量的东西现在就成了这个类的数据成员。程序的 main 函数简单地创建这个假类的一个对象并调用这个类中的一个“主”方法。从技术上说，程序确实没有使用全局变量，但假类使得程序仍然具有原来该有的那些缺陷。

另一种类型的假类是因为程序员觉得面向对象编程总是“更好”，并在事实并不适用类的情况下也创建了类。在这类情况下，程序员常常创建一个封装了非常特定功能的类，它

只在编写原先程序的环境中才是合理的。有两种方法可以测试是否编写了这种类型的假类。第一种是提出问题“我是否可以给这个类取个更具体并且更短的名称？”如果发现一个像PayrollReport ManagerAndPrintSpooler这样的类名，很可能就遇到了这种假类。

另一种测试是提问“如果我打算编写另一个具有相似功能的程序，自己是否可以想象怎样复用这个类，最多只需要一些微小的修改？还是需要极大的变动？”

即使在C++中，还是不可避免会出现一些假类。例如，我们必须封装数据以便在集合类中使用。但是，这样的类通常较小并且都是很基本的类。如果我们可以避免复杂的假类，就可以提高代码的质量。

5.5.2 单功能

如果读者曾经看过电视节目《美味佳肴》，应该知道主持人Alton Brown花了大量的时间讨论应该怎样装备厨房，使之发挥最大的效用。他常常抱怨那些被他称为单功能的厨房器具，批评它们只能完成一项任务而不能干其他事情。在编写类的时候，我们应该尽量使它们变得通用，同时又包含程序所需要的所有特定功能。

实现这个目的的一个方法是使用模板类。这是个高级主题，其语法稍稍有些晦涩。但是，它允许我们所创建类的一个或多个数据成员的类型在这个类的对象被创建时才指定。模板类允许我们“重构”基本的功能。例如，我们的`studentCollection`类包含了大量的对于封装了链表的任何类都适用的代码。我们可以为基本的链表创建一个模板类，这样链表节点所包含的数据的类型就可以在创建这个模板类的对象时再指定，而不是固定为`studentRecord`。接着，我们的`studentCollection`类将拥有一个模板链表类的对象作为它的数据成员，而不是一个链表头指针，因而不再需要对链表进行直接的操作。

模板类超出了本书的范围。但是，作为类设计人员，如果想发展自己的技能，应该尽量使类具有多功能。发现一个当前问题可以用很久以前（远早于知道当前这个问题的存在）所编写的一个类解决时，那感觉无疑会棒极了。

5.6 习题

5.1 让我们尝试用基本框架实现一个类。考虑用一个类存储表示一辆汽车的数据。这个类需要保存3个数据：用字符串表示的厂商名和型号名以及用整数表示的型号年份。为每个数据成员创建一对get/set方法。对于像成员名称这样的细节做出良

好的决定。读者并不一定要遵循我所采用的特定命名约定。重要的是决定了自己所做的选择之后就要保持一致风格。

5.2 对于前一个例子的汽车类，添加一个支持方法，它以一个格式化字符串的形式返回一个汽车对象的完整描述，例如，“1957 Chevrolet Impala”。添加另一个支持方法，返回用年数表示的车龄。

5.3 采用第 4 章的可变长度的字符串函数（append、concatenate 和 characterAt）创建一个表示可变长度的字符串的类，确保实现所有必要的构造函数、一个析构函数和一个重载的赋值操作符。

5.4 对于前一个习题的可变长度的字符串类，用重载的[]操作符替代**characterAt**方法。例如，如果 **myString** 是这个类的一个对象，则 **myString[1]**应该返回与**myString.characterAt(1)**相同的结果。

5.5 对于前面习题的可变长度的字符串类，添加一个 remove 方法，它接受一个起始位置和需要从字符串的中间删除的字符数量为参数。因此，myString.remove(5,3)将删除从第 5 个位置开始的 3 个字符。保证这个方法在任一参数为非法值的情况下仍然能够表现出适当的行为。

5.6 对可变长度的字符串类进行研究，看看有没有可能对它进行重构。例如，是否存在任何公共的功能可以分离到一个私有的支持方法？

5.7 用第 4 章所描述的学生记录函数（addRecord 和 averageRecord）创建一个表示一个学生记录集合的类。和以前一样，保证实现所有必要的构造函数、一个析构函数和一个重载的赋值操作符。

5.8 对于前一个习题的学生记录集合类，添加一个 Recordswithin Range 方法，它接受一个低的成绩和一个高的成绩为参数，并返回一个新的学生记录集合，其中的记录位于这两个成绩的范围之内（原先的集合不会受到影响）。例如，myCollection.RecordsWithin Range(75, 80)将返回一个成绩范围在 75～80（包括这两个成绩）之间的所有记录的集合。

第 6 章

用递归解决问题

本章将讨论递归，也就是一个函数直接或间接调用自身的问题。

递归编程看上去很简单，实质也的确如此。事实上，良好的递归解决方案往往非常简单优雅。但是，通向解决方案的道路却并非如此。这是因为递归要求我们采用与其他编程不同的思路。当使用循环处理数据时，我们会考虑用线性的方法进行处理。但是，当我们用递归处理数据时，常规的线性思考过程就没什么用武之地。许多羽翼已丰的优秀程序员在面临递归问题时常常束手无策，因为他们无法把已经学会的问题解决技巧应用于递归问题。在本章中，我们将讨论怎样系统性地解决递归问题。关键在于使用我们所称的“大递归思路”，或缩写为 BRI（Big Recursive Idea）。这是一种相当简单明了的思路，尽管看上去似乎有点奇怪，但它确实可行。

6.1 递归基础知识回顾

关于递归的语法，并没有太多需要说明的。困难在于怎样用递归解决问题。当一个函

数在任意时候调用自身时就发生了递归，因此递归的语法就是函数调用的语法。最常见的形式是直接递归，就是对函数的调用发生在同一个函数的函数体中。例如：

```
int factorial(int n) {
  ❶if (n == 1) return 1;
   else return n * ❷factorial(n - 1);
}
```

这个函数用于计算 n 的阶乘，它是一个常见的演示递归的例子，但它的效率却非常低下。例如，当 n 为 5 时，它的阶乘就是从 5 到 1 之间的所有整数的乘积，即 120。注意，在有些情况下不会发生递归。在这个函数中，如果参数的值为 1，这个函数不需要任何递归就简单地返回这个值❶。这被称为基本情况。在其他情况下，需要进行递归调用❷。

另一种形式的递归是间接递归。例如，如果函数 A 调用了函数 B，而后者后来又调用了函数 A。间接递归很少作为问题解决技巧使用，因此我们不打算对它进行讨论。

6.2 头递归和尾递归

在讨论大递归思路之前，我们需要理解头递归和尾递归的区别。在头递归中，递归发生在函数的其他处理代码之前（可以把它看成是发生在函数的头部或顶部）。在尾递归中，情况正好相反。函数的其他处理代码发生在递归调用之前。在这两种风格的递归之间做出选择看上去似乎是随意的，但这种选择可能会导致截然不同的结果。为了说明它们之间的区别，让我们观察两个问题。

多少只鹦鹉

热带天堂铁路（TRP）上的乘客渴望能够透过车窗看到一群色彩斑斓的鹦鹉。因此，铁路方对维护当地鹦鹉群的健康极为关注，并决定记录铁路主线的每个站台上能够看到的鹦鹉的数量。每个站台由一位 TPR 员工负责（如图 6.1 所示），由他负责鹦鹉的计数。遗憾的是，这项本应该很简单的任务却由于铁路方所使用的电话系统的简陋而变得非常复杂。每个站台只能与直接相邻的站台通话。我们怎么才能在铁路的终点站得知鹦鹉总数呢？

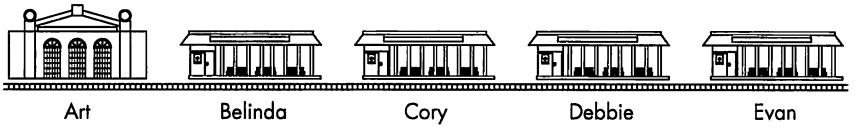

图 6.1 5 个站点的员工只能与直接相邻的站点通话

我们假设作为 Art 所在的铁路主线终点车站共有 7 只鹦鹉，Belinda 所在车站有 5 只鹦鹉，Cory 所在车站有 3 只鹦鹉，Debbie 所在车站有 10 只鹦鹉，Evan 所在的最后一个车站有 2 只鹦鹉。因此，鹦鹉的总数是 27。问题在于，员工们怎样互相协作把这个总数传递给位于终点站的 Art 呢？这个问题的任何解决方案都需要在铁路主线两端之间来回通话。每个车站的员工需要对鹦鹉进行计数并汇报他的观察结果。即使是这样，仍然有两种不同的方法来完成这条通信链，这两种方法分别对应于编程中的头递归和尾递归技巧。

方法 1

在这种方法中，当我们沿着向外的通信线路进行接力时，保存到目前为止的鹦鹉总数。每名员工向铁路沿线的下一个站点发出通话请求，把到目前为止所看到的鹦鹉数量传递给对方。到达线路的一端时，Evan 将是第 1 个得知鹦鹉总数的站点员工，然后他把这个信息汇报给 Debbie，后才再汇报给 Cory，接下来依次类推（如图 6.2 所示）。

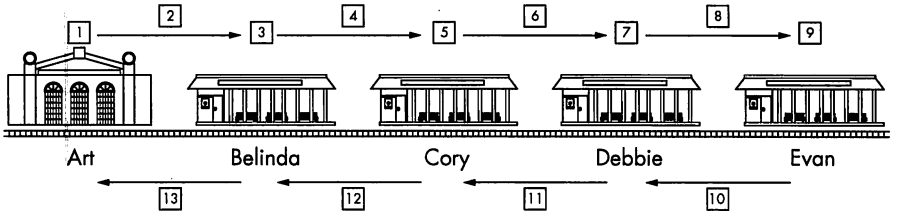

图 6.2 鹦鹉计数问题的方法 1 所采取的计数步骤

1. Art 首先计数他的站台的鹦鹉，计数的结果是 7 只鹦鹉。
2. Art 向 Belinda 通话：“终点站共有 7 只鹦鹉。”
3. Belinda 数出本站点共有 5 只鹦鹉，并算出目前的鹦鹉总数为 12 只。
4. Belinda 向 Cory 通话：“前两个站点共有 12 只鹦鹉。”
5. Cory 计数的结果是 3 只鹦鹉。
6. Cory 向 Debbie 通话：“前 3 个站点共有 15 只鹦鹉。”
7. Debbie 数出 10 只鹦鹉。
8. Debbie 向 Evan 通话：“前 4 个站点共有 25 只鹦鹉。”
9. Evan 数出 2 只鹦鹉，并发现鹦鹉的总数是 27 只。

10. Evan 向 Debbie 通话："鹦鹉的总数是 27 只。"

11. Debbie 向 Cory 通话："鹦鹉的总数是 27 只。"

12. Cory 向 Belinda 通话："鹦鹉的总数是 27 只。"

13. Belinda 向 Art 通话："鹦鹉的总数是 27 只。"

这种方法类似于尾递归。在尾递归中，递归调用发生在处理之后，即递归调用是函数的最后一个步骤。在上面的通信链中，注意员工的"工作"（即对鹦鹉进行计数并汇总）发生在他们向沿线的下一个站点发出信号之前。所有的工作都是在外向的通信链（即从终点站 Art 到 Evan）中完成的，而不是在内向的通信链（即从 Evan 到 Art）中完成的。下面是每位员工所采取的步骤：

1. 对站台上可以看到的鹦鹉数量进行计数。

2. 把这个数量与前一个站点所提供的总数相加。

3. 与下一个站点通话，汇报当前的鹦鹉总数。

4. 等待下一个站点向他汇报鹦鹉总数，然后把这个总数传递给前一个站点。

方法 2

在这种方法中，我们从另一个方向计数鹦鹉的总数。每个员工与铁路上的下一个站点联系时，向该车站请求鹦鹉总数。然后这位员工把这个鹦鹉总数与自己站点上的鹦鹉总数相加，并把总数传给铁路沿线的下一个站点（如图 6.3 所示）。

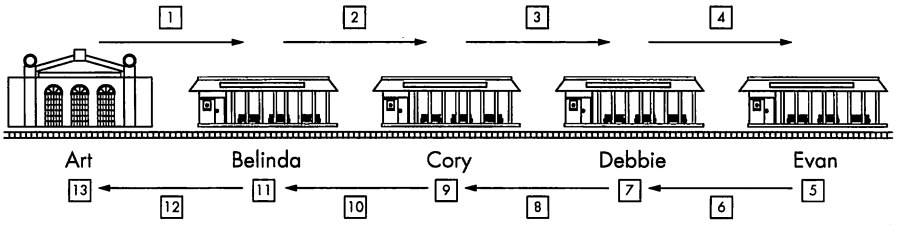

图 6.3 鹦鹉计数问题的方法 2 所采取的计数步骤

1. Art 与 Belinda 通话："到你的站点为止的鹦鹉总数是多少？"

2. Belinda 与 Cory 通话："到你的站点为止的鹦鹉总数是多少？"

3. Cory 与 Debbie 通话："到你的站点为止的鹦鹉总数是多少？"

4. Debbie 与 Evan 通话："到你的站点为止的鹦鹉总数是多少？"

5. Evan 是最后一个站点，他数出 2 只鹦鹉。

6. Evan 与 Debbie 通话："到本站点为止的鹦鹉总数是 2 只。"

7. Debbie 在她的站点数出 10 只鹦鹉，因此到她的站点为止的鹦鹉总数是 12 只。

8. Debbie 与 Cory 通话："到本站点为止的鹦鹉总数是 12 只。"

9. Cory 数出 3 只鹦鹉。

10. Cory 与 Belinda 通话："到本站点为止的鹦鹉总数是 15 只。"

11. Belinda 数出 5 只鹦鹉。

12. Belinda 与 Art 通话："到本站点为止的鹦鹉总数是 20 只。"

13. Art 在终点站数出 7 只鹦鹉，从而得出鹦鹉总数为 27 只。

这种方法类似于头递归。在头递归中，递归调用发生在其他处理代码之前。在这个例子中，先与下一个车站通话，然后再计数鹦鹉数量并计算总和。"工作"被推迟到沿线的站点汇报了他们的总数之后。下面是每位员工所采取的步骤：

1. 与下一个站点通话。

2. 对当前站点可见的鹦鹉数量进行计数。

3. 把这个数量与下一个站点所提供的鹦鹉总数相加。

4. 把相加所得的结果汇报给前一个站点。

读者很可能已经注意到了这两种不同方法的实际效果。在第 1 种方法中，最终所有的站点员工都知道了鹦鹉的总数。在第 2 种方法中，只有终点站 Art 知道鹦鹉的总数。但是，Art 也是唯一需要知道总数的员工。

当我们把上面这些讨论结果转换为程序代码时，另一个实际效果对于我们的分析则显得更为重要。在第 1 种方法中，每个员工在发出请求时把"当前总数"汇报向铁路沿线的下一个站点。在第 2 种方法中，员工们简单地向下一个站点请求信息，而不需要沿线传递任何数据。这是头递归方法的典型效果。由于递归调用是首先发生的，出现在任何其他处理代码之前，因此没有什么新信息可以提供给递归调用。一般而言，头递归允许向递归调

用传递尽可能少的数据。现在，让我们观察另一个问题。

谁是我们的最佳顾客

DelegateCorp 公司的经理需要确定 8 位顾客中哪位能够为公司带来最大收益。有两个因素使这个看上去简单的任务变得有点复杂。首先，为了确定一位顾客的总收入，需要检查这位顾客的所有档案并记录数十张订单和收据。其次，DelegateCorp 公司的员工就像这家公司的名称所提示的那样喜欢委托，只要有可能总是尽量把工作移交给下一级别的某位员工。为了使情况尽在掌握，经理制定了一个规则：在进行委托时，必须亲自完成某部分的工作，并且委托别人所完成的工作量必须小于自己所接受的工作量。

表 6.1 和表 6.2 确认了 DelegateCorp 公司的员工和顾客。

表 6.1 DelegateCorp 公司的员工头衔和等级

头衔	等级
经理	1
常务副经理	2
副经理	3
经理助理	4
初级经理	5
实习生	6

表 6.2 DelegateCorp 公司的顾客

顾客编号	收益
#0001	$172 000
#0002	$68 000
#0003	$193 000
#0004	$13 000
#0005	$256 000
#0006	$99 000

根据公司对于委托工作的规定，下面是针对 6 位顾客的档案将要发生的事情。经理亲自查看一份档案并确定该顾客为公司创造了多少收益。经理将把其他 5 份档案委托给常务

副经理。常务副经理亲自处理其中一份档案并把其余 4 份档案委托给副经理。这个过程将一直持续，直到把任务传递给第 6 位员工也就是实习生，他总共需要处理 1 份档案，并且必须亲自完成，不能再委托其他人处理。

图 6.4 所示描述了通信链和劳动力的细分。但是，和前一个例子一样，两条通信链采取了截然不同的方法。

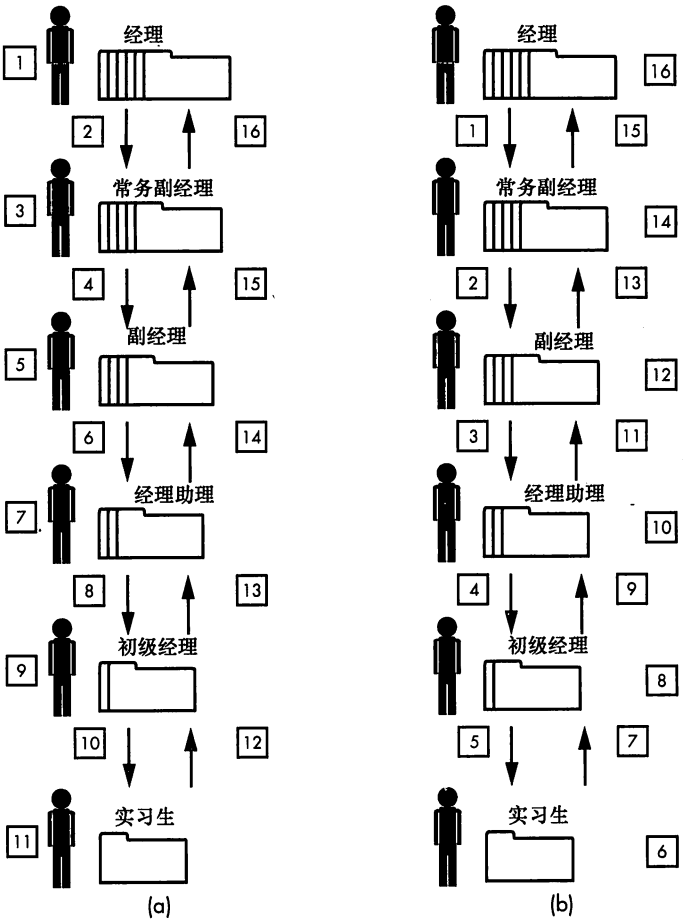

图 6.4 寻找最大收益顾客的方法 1（a）和方法 2（b）的步骤编号

方法 1

在这种方法中，在委托剩余的档案时，员工还传递到目前为止的最高收益总额。这意味着员工必须记录一份档案的收益，然后把它与之前所看到的最高数额进行比较，然后再

把剩余的档案委托给其他员工。下面是这种方法实际操作的一个例子：

1. 经理记录顾客#0001 的收益，其金额为$172 000。
2. 经理传达给常务副经理："到目前为止的最大金额是$172 000，来自顾客#0001。把剩下的 5 份档案拿去，并确定最高的整体收益。"
3. 常务副经理记录顾客#0002 的收益，其值为$68 000。到目前为止的最高收益仍然是顾客#0001 的$172 000。
4. 常务副经理传达给副经理："到目前为止的最大金额是$172 000，来自顾客#0001。把剩下的 4 份档案拿去，并确定最高的整体收益。"
5. 副经理记录顾客#0003 的收益，其值为$193 000。到目前为止的最高收益是顾客#0003 的$193 000。
6. 副经理传达给经理助理："到目前为止的最大金额是$193 000，来自顾客#0003。把剩下的 3 份档案拿去，并确定最高的整体收益。"
7. 经理助理记录顾客#0004 的收益，其值为$13 000。到目前为止的最高收益是顾客#0003 的$193 000。
8. 经理助理传达给初级经理："到目前为止的最大金额是$193 000，来自顾客#0003。把剩下的 2 份档案拿去，并确定最高的整体收益。"
9. 初级经理记录顾客#0005 的收益，其值为$256 000。到目前为止的最高收益是顾客#0005 的$256 000。
10. 初级经理传达给实习生："到目前为止的最大金额是$256 000，来自顾客#0005。把最后一份档案拿去，并确定最高的整体收益。"
11. 实习生记录顾客#0006 的收益，其值为$99 000。到目前为止的最高收益是顾客#0005 的$256 000。
12. 实习生向初级经理汇报："所有顾客的最高收益是顾客#0005 的$256 000。"
13. 初级经理向经理助理汇报："所有顾客的最高收益是顾客#0005 的$256 000。"
14. 经理助理向副经理汇报："所有顾客的最高收益是顾客#0005 的$256 000。"
15. 副经理向常务副经理汇报："所有顾客的最高收益是顾客#0005 的$256 000。"

16. 常务副经理向经理汇报："所有顾客的最高收益是顾客#0005 的$256 000。"

图 6.4（a）所显示的这种方法使用了尾递归技巧。每个员工处理一份档案文件并把对该顾客进行计算所产生的收益与目前为止的最大收益进行比较。然后，这位员工把比较结果传送给下属员工。递归（也就是剩余工作的传递）发生在其他处理之后。每个员工的处理过程类似下面这样：

1. 记录一份顾客档案的收益。
2. 把这份收益总和与上级领导在其他顾客档案中所看到的最大收益进行比较。
3. 把剩余的顾客档案传达给一位下属员工，同时向他传达到目前为止的最大收益金额。
4. 当下属员工返回所有顾客档案的最大收益金额时，把这个金额汇报给上级领导。

方法 2

在这种方法中，每位员工首先留下一份顾客档案，然后把剩余的档案移送给下属员工。在这种情况下，下属员工并不知道所有档案的最大收益，而是只能看到下属员工所提供的当前最大金额。和第一个示例问题一样，这种方法简化了请求。使用与第 1 种方法相同的数据，对话将变成：

1. 经理传达给常务副经理："把这 5 份档案拿去，告诉我最高的收益金额。"
2. 常务副经理传达给副经理："把这 4 份档案拿去，告诉我最高的收益金额。"
3. 副经理传达给经理助理："把这 3 份档案拿去，告诉我最高的收益金额。"
4. 经理助理传达给初级经理："把这 2 份档案拿去，告诉我最高的收益金额。"
5. 初级经理传达给实习生："把这份档案拿去，告诉我最高的收益金额。"
6. 实习生记录顾客#0006 的收益，其值为$99 000。这是实习生唯一能够看到的档案，因此它是到目前为止的最高收益。
7. 实习生向初级经理汇报："我处理的档案的最高收益是顾客#0006 的$99 000。"
8. 初级经理记录顾客#0005 的收益，其值为$256 000。这位员工所知道的最高收益是顾客#0005 的$256 000。
9. 初级经理向经理助理汇报："我所接手的档案中的最高收益是顾客#0005 的$256 000。"

10. 经理助理记录顾客#0004 的收益，其值为$13 000。这位员工所知道的最高收益是顾客#0005 的$256 000。
11. 经理助理向副经理汇报："我所接手的档案中的最高收益是顾客#0005 的$256 000。"
12. 副经理记录顾客#0003 的收益，其值为$193 000。这位员工所知道的最高收益是顾客#0005 的$256 000。
13. 副经理向常务副经理汇报："我所接手的档案中的最高收益是顾客#0005 的$256 000。"
14. 常务副经理记录顾客#0002 的收益，其值为$68 000。这位员工所知道的最高收益是顾客#0005 的$256 000。
15. 常务副经理向经理汇报："我所接手的档案中的最高收益是顾客#0005 的$256 000。"
16. 经理记录顾客#0001 的收益，其值为$172 000。他所知道的最高收益是顾客#0005 的$256 000。

图 6.4（b）显示了这种方法，它使用了头递归技巧。每位员工仍然必须记录一份顾客档案的收益，但具体的操作延迟到下属员工确定了剩余档案的最大收益金额之后。每位员工的处理过程介绍如下。

1. 给自己留下一份档案之后，把剩余的顾客档案移送给一位下属员工。
2. 从这位下属员工那里获取这些顾客档案的最高收益金额。
3. 记录自己所留下的那份顾客档案的收益。
4. 把两个收益值中较大的那个汇报给上级领导。

和"鹦鹉计数"问题一样，头递归技巧允许每位员工向下属员工传递尽可能少的信息。

6.3 大递归思路

现在我们开始解释大递归思路。事实上，如果读者已经理解了上面的示例问题的步骤，实际上已经领略了大递归思路。

大递归思路到底是怎么样的呢？这两个示例问题都采用了递归解决方案的形式。通信链中的每个人对最初数据的一个更小子集执行相同的步骤。但是，值得注意的是，问题本

身并没有涉及到递归。

在第 1 个问题中，每位铁路员工向沿线的下一个站点发出一个请求，而下一位员工在满足这个请求时采取了与前面这位员工相同的步骤。但是在请求中并未要求任何一位员工采取这些特定的步骤。例如，当方法 2 中 Art 与 Belinda 通话时，他要求她对从她的车站开始直到线路终点的所有站点的鹦鹉进行计数。他并没有提示用什么方法来计算总数。如果他考虑到了这一点，可能会意识到 Belinda 所采取的步骤实际上与他相同。但是，他并不需要考虑这一点。为了完成这个任务，Art 所需要的只是让 Belinda 提供他所询问的问题的正确答案。

类地，在第 2 个问题中，管理链中的每位员工都尽可能多地把工作留给下属完成。例如，经理助理可能对初级经理非常了解，预期初级经理会把大部分档案留给实习生处理。但是，经理助理并没有理由关心初级经理是亲自处理所有的档案还是把大部分留给下属处理。

经理助理只关心初级经理返回了正确的答案。由于经理助理并不打算重复初级经理所完成的工作，因此他简单地假设初级经理所返回的答案是正确的，并使用这个数据来解决副经理委托给他的任务。

在这两个问题中，当员工们向其他员工发送请求时，他们所关注的是结果而不是过程。每位员工移交了一个问题并收到一个答案。这正是大递归思路：如果在编写代码时采用某种约定，可以假装并没有发生递归。我们甚至可以使用一种欺骗手段（如下所示）把一种迭代式实现转移到一个递归式实现，而不需要考虑递归是怎样解决问题的。随着时间的推移，我们会对递归解决方案的工作原理有个直观的理解，但是在获得这种直觉之前已经能够在代码中创建递归式的实现并对它的效果充满信心。

让我们通过一个代码例子把这些概念应用于实践中。

计算一个整数数组的和

编写一个递归函数，接受一个整数数组和该数组的长度为参数。该函数返回这个数组中各个整数的和。

读者的第一个想法可能是这个问题如果用迭代的方式解决将很简单。事实上，我们首先观察这个问题的一种迭代式解决方案：

```
int iterativeArraySum(int integers[], int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += integers[i];
    }
    return sum;
}
```

这段代码与第 3 章的代码非常相似，因此这个函数应该非常容易理解。下一个步骤是编写代码，作为迭代解决方案和最终所需要的递归解决方案的过渡。我们将保留这个迭代函数并添加第 2 个称为调度器的函数。调度器把大部分工作移交给一个以前编写的迭代函数，并用它所返回的信息来解决整体问题。为了编写一个调度器，我们必须遵循两个规则：

1. 调度器必须完整地处理最简单的情况，而无需再调用迭代函数；

2. 当调度器调用迭代函数时，必须向它传递问题的更简单版本。

把第 1 个规则应用于这个问题时，我们必须确定什么是最简单情况。如果长度为 0，从概念上说相当于向这个函数传递了一个“空”数组，其元素之和就为 0。人们可能觉得最简单情况应该是长度为 1 的时候。在这种情况下，参数数组中只有 1 个数，我们可以把这个数作为总和返回。这两种解释都是合理的，但是第 1 种选择允许这个函数处理数组长度为 0 这种特殊情况。注意，原先的迭代函数在长度为 0 时并不会失败，因此值得维持这种灵活性。

为了把第 2 个规则应用于这个问题，我们必须想出一种方法，把一个更小版本的问题从调度器函数传递给迭代函数。没有简单的办法传递一个更简单的数组，但我们可以很方便地传递一个更小的长度值。如果调度器所传递的长度值为 10，相当于要求迭代函数计算数组中 10 个值的和。如果调度器函数向迭代器传递 9 作为长度参数的值，相当于要求迭代函数计算数组中前 9 个值的和。然后，调度器函数可以加上数组中剩余的一个值（第 10 个值）来计算所有 10 个值的和。注意，在调用迭代函数时将长度减 1 最大限度地发挥了迭代函数的作用，从而使调度器函数的工作量减到最低。就像 DelegateCorp 公司的经理们一样，这正是我们所需要的方法，就是调度器尽可能地减少自己的工作量。

把这些思路汇集在一起，就可以实现这个问题的调度器函数：

```
int arraySumDelegate(int integers[], int size) {
  ❶if (size == 0) return 0;
  ❷int lastNumber = integers[size - 1];
    int allButLastSum = ❸iterativeArraySum(integers, size - 1);
  ❹return lastNumber + allButLastSum;
}
```

第 1 条语句实行了调度器的第 1 个规则：它检查最简单情况并对其进行完全处理。在这种情况下，它返回 0❶。否则，控制转移到剩余的代码，实行第 2 个规则。数组中的最后一个数被存储到一个称为 lastNumber 的局部变量中❷，然后通过调用迭代函数计算数组中其余值的总和❸。这个调用所返回的结果存储在另一个称为 allButLastSum 的局部变量中，最后这个函数返回这两个局部变量之和❹。

如果我们正确地创建了一个调度器函数，就已经有效地创建了一个递归解决方案，这就是实际使用中的大递归思路。为了把这个迭代式解决方案转移为递归式解决方案，只需要一个非常简单的步骤：在以前调用迭代函数的时候，让委托函数调用自身。现在，我们就可以删除迭代函数了。

```
int ❶arraySumRecursive(int integers[], int size) {
    if (size == 0) return 0;
    int lastNumber = integers[size - 1];
    int allButLastSum = ❷arraySumRecursive(integers, size - 1);
    return lastNumber + allButLastSum;
}
```

以上的代码只需要进行两处修改。对函数的名称进行了修改，以更好地描述它的新形式❶。另外，这个函数在以前调用迭代函数的地方改为调用自身❷。arraySumDelegate 和 arraySumRecursive 这两个函数的逻辑是相同的。每个函数检查总和已知的最简单情况（在此例中，就是数组的长度为 0，其和为 0）。如果不是最简单情况，每个函数就调用自身，要求其计算数组中除最后一个值之外所有值的和。最后，每个函数把返回的和与数组中的最后一个值相加得到总和。唯一的区别是第 1 个版本的函数调用了另一个函数，而递归版本的函数调用了自身。大递归思路告诉我们，如果遵循了编写调度器函数的两个规则，就可以忽略这个区别。

在采用大递归思路时，并不需要完全遵守上面所显示的步骤。具体地说，在实现递归解决方案之前，通常并不需要实现问题的迭代解决方案。编写一个迭代函数作为垫脚石是项额外的工作，最终将被抛弃。除此之外，递归解决方案最好应用于难以应用迭代解决方案的场合，其理由稍后解释。但是，在不用实际编写迭代解决方案的情况下也可以采用大递归思路的计划。关键在于把递归调用看成是对另一个函数的调用，而不需要关注这个函数的内部细节。按照这种方式，可以从递归解决方案中消除递归逻辑的复杂性。

6.4 常见的错误

如上所述，只要采用正确的方法，递归解决方案往往非常容易编写。但是，如果不小

心，很容易出现不正确的递归实现或者递归解决方案，虽然能够“工作”但得不偿失。递归实现的大部分错误来自两种基本错误：过度思考问题或者在开始实现时没有清晰的计划。

过度思考递归问题对于程序员新手尤为常见，由于经验有限并且缺乏信心，导致他们把问题想象得比实际上更为困难。由于过度思考所产生的代码可以通过它过于细致的外观而被识别。例如，一个递归函数在只需要考虑一种特殊情况下实际可能考虑了好几种特殊情况。

太早开始实现可能导致过于复杂的“Rube Goldberg（传说中的一种过分复杂的装置）”代码，无法预见的交互导致源代码到处需要修改。

让我们观察几个特定的错误，并讨论怎么避免它们。

6.4.1 过多的参数

如前所述，头递归技巧可以减少向递归调用所传递的数据量，而尾递归可能导致向递归调用传递额外的数据。程序员常常陷入到尾递归模式上，因为他们过度思考了问题并且太急于开始实现。

考虑以递归方式计算整数数组之和的问题。为这个问题编写一个迭代解决方案时，程序员知道需要一个“到目前为止的和”变量（在前面所提供的迭代解决方案中，这个变量称为 sum），并且知道这个数组将从第 1 个元素开始求和。考虑递归解决方案时，程序员很自然地想到一种绝大部分内容直接抄录迭代解决方案的实现，保留了表示目前为止总和的变量以及处理数组中第 1 个元素的第 1 个递归调用。但是，这种方法要求递归函数传递到目前为止的和以及下一个递归调用应该开始处理的位置。这种解决方案如下所示：

```
int arraySumRecursiveExtraParams(int integers[], int size, ❶int sum, ❷int currentIndex) {
    if (currentIndex == size) return sum;
    sum += integers[currentIndex];
    return arraySumRecursiveExtraParameters(integers, size, sum, currentIndex + 1);
}
```

这段代码与其他递归版本一样短，但它在语义上却复杂了很多，因为多了额外的参数 sum❶和 currentIndex❷。站在客户代码的角度，这两个额外的参数是没有意义的，它们在调用中的值始终是 0，如下所述：

```
int a[10] = {20, 3, 5, 22, 7, 9, 14, 17, 4, 9};
int total = arraySumRecursiveExtraParameters(a, 10, 0, 0);
```

这个问题可以通过使用包装器函数来避免，具体在下一节描述。但是，由于我们无法

彻底消除这些参数，因此它并不是最佳解决方案。这个迭代函数和原来的递归函数所回答的问题是：“有这些数量的元素的数组的和是多少？”反之，第 2 个递归函数需要回答的问题是：“如果这个数组正好有这些数量的元素，那么这个数组的和是多少？我们从这个特定的元素（第 3 个参数）开始处理，并且这（最后一个参数）是之前所有元素的和。”

为了避免“过多的参数”问题，可以在考虑递归之前就选择函数的参数。换句话说，迫使自己使用与迭代解决方案相同的参数列表。如果使用完整的大递归思路过程，首先实际编写迭代函数，就可以自动避免这个问题。但是，如果跳过了完整的过程，仍然可以从概念上使用这个过程，根据迭代函数期望接受的参数列表来编写递归函数的参数列表。

6.4.2 全局变量

在避免过多的参数时，有时候会导致程序员犯下一个不同的错误：使用全局变量从一个递归调用向另一个递归调用传递数据。使用全局变量通常是不良的编程实践，尽管有时候由于性能原因允许这种做法。在递归函数中，只要有可能，应该尽量避免使用全局变量。我们可以观察一个特定的问题，看看程序员是怎样犯下这种错误的。假设要求我们编写一个递归函数，对一个整数数组中所出现的零的数量进行计数。这个问题很容易用迭代方法来解决：

```
int zeroCountIterative(int numbers[], int size) {
   int sum = 0;
 ❶int count = 0;
   for (int i = 0; i < size; i++) {
      if (numbers[i] == 0) count ++;
   }
   return count;
}
```

这段代码的逻辑非常清晰简单。我们从数组的第 1 个位置开始处理，对零的数量进行计数，并使用一个局部变量 `count` 作为计数器❶，直到处理完最后一个位置。但是，如果我们在编写递归函数时头脑里已经有了一个类似这样的函数，可能会认为这个递归函数也需要一个表示计数器的变量。我们无法简单地在递归函数中把 `count` 声明为局部变量，因为它在每次递归调用内部都会变成一个新变量。因此，我们可能会想到把它声明为全局变量：

```
int count;
int zeroCountRecursive(int numbers[], int size) {
   if (size == 0) return count;
   if (numbers[size - 1] == 0) count++;
   zeroCountRecursive(numbers, size - 1);
}
```

这段代码可以达到目的，但这个全局变量是完全没有必要的，会导致全局变量在典型情况下可能发生的所有问题，例如代码的不容易理解和不容易维护等。有些程序员可能想到把这个变量声明为静态局部变量来消除这个问题：

```
int zeroCountStatic(int numbers[], int size) {
 ❶static int count ❷= 0;
   if (size == 0) return count;
   if (numbers[size - 1] == 0) count++;
   zeroCountStatic(numbers, size - 1);
}
```

在 C++中，当一个局部变量被声明为静态变量时，它在不同的函数调用之间能够保留原值。因此，这个局部静态变量 count❶所发挥的作用与前一个版本中的全局变量相同。那么，它存在什么问题吗？这个变量被初始化为 0❷，这是在这个函数第 1 次调用时发生的。这对于静态变量而言是必要的，否则它就没什么用处。但是，这意味着这个函数只在第 1 次被调用时才会返回正确的值。如果这个函数被调用了 2 次，第 1 次接受了一个包含 3 个 0 的数组，第 2 次接受了一个包含 5 个 0 的数组。对于第 2 个数组，这个函数将返回 8，因为当第 2 次调用开始时，count 仍将保留它为第 1 次调用结束时的值。

在这个例子中，避免使用全局变量的解决方案就是使用大递归思路。我们可以假设一个接受更小长度值的递归调用将返回正确的结果，并且可以通过它的返回值计算数组的正确结果。这就形成了一种头递归解决方案：

```
int zeroCountRecursive(int numbers[], int size) {
   if (size == 0) return 0;
 ❶int count = zeroCountRecursive(numbers, size - 1);
 ❷if (numbers[size - 1] == 0) count++;
 ❸return count;
}
```

在这个函数中，仍然有一个局部变量 count❶，但是没有在不同的调用之间保留它的原值。反之，它存储了递归调用的返回值，我们可以选择在返回这个变量❸之前把它的值增加 1❷。

6.5 把递归应用于动态数据结构

递归常常应用于像链表、树和图这样的动态数据结构。数据结构越复杂，递归解决方案在简化代码方面所发挥的作用也就越大。处理复杂的数据结构常常类似于在迷宫中寻找

一条正确的出路，而递归允许我们在处理过程中回溯到以前步骤。

6.5.1 递归和链表

但是，我们首先还是从最基本的动态数据结构也就是链表开始。对于本节的讨论，假设我们所使用的链表是最简单的节点结构，即每个节点的数据只是一个整数。链表的类型声明如下所示：

```
struct listNnode {
    int data;
    listNode * next;
};
typedef listNode * listPtr;
```

把大递归思路应用于单链表时遵循了相同的基本计划，与特定的任务无关。递归要求我们对问题进行细分，把原问题的一个削减版本传递给递归调用。只有一种可行的方式可以细分这个单链表，那就是把它分为链表的第 1 个节点以及链表的剩余节点。

在图 6.5 中，我们看到一个示例链表被划分为 2 个不相等的部分：第 1 个节点和所有其余节点。从概念上说，我们可以把原链表的“剩余部分”看成是一个单独的链表，以原链表的第 2 个节点作为它的第 1 个节点。站在这个角度，就可以很顺利地实现递归。

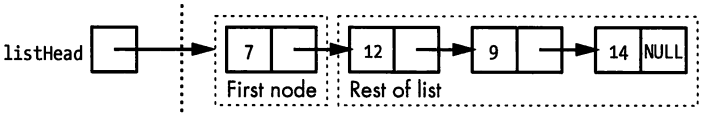

图 6.5 一个链表被划分为第 1 个节点和“链表的剩余部分”

但是，为了实现递归，我们并不需要描绘出所有的递归步骤。站在编写用递归函数处理链表的程序员的角度，从概念上说把可以链表分为两个部分：必须处理的第 1 个节点以及不需要处理的链表的剩余部分，因此不需要对后者加以关注。这是图 6.6 所显示的态度。

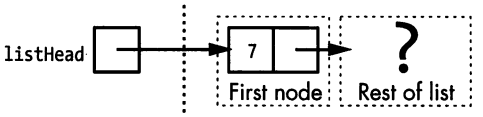

图 6.6 当程序员使用递归时应该把链表想像成两个部分：第 1 个节点以及可以作为混沌的形状传递给递归调用的链表剩余部分

确定了工作的划分之后，我们认为单链表的递归处理可以按照下面的通用计划来进行。假设有一个链表 L 和一个问题 Q：

1. 如果 L 是最小情况，我们就直接设置一个默认值。否则……
2. 使用一个递归调用为链表 L 的“剩余部分”（从 L 的第 2 个节点开始）产生 Q 的答案。
3. 检查 L 的第 1 个节点的值。
4. 使用前两个步骤的结果为整体的 L 产生 Q 的答案。

正如我们所看到的那样，对链表的划分施加了实际限制之后，这就是对大递归思路的一种简单应用。现在我们把这个蓝图应用于一个特定的问题。

对一个单链表中的负数进行计数

编写一个递归函数，它对一个单链表（元素的数据类型为整数）进行处理。这个函数返回这个链表中负数的数量。

我们需要回答的问题 Q 是：这个链表中有多少个负数？因此，我们的计划可以陈述为：

1. 如果这个链表没有任何节点，计数就默认为 0。否则……
2. 使用一个递归调用对链表的“剩余部分”中的负数进行计数。
3. 判断链表的第 1 个节点的值是否为负数。
4. 使用前两个步骤的结果确定整个链表中的负数数量。

下面是一个直接实现了上面这个计划的函数实现：

```
int countNegative(listPtr head) {
    if (head == NULL) return 0;
    int listCount = countNegative(head->next);
    if (head->data < 0) listCount++;
    return listCount;
}
```

注意，这段代码遵循了与前面的例子相同的原则。它“反向”地对负数进行计数，从链表的尾部开始直到链表的头部。另外，注意这段代码采用了头递归技巧。我们在处理第 1 个节点之前处理链表的“剩余”部分。和此前一样，它允许我们避免在递归调用中传递额

外的数据或者使用全局变量。

另外，注意链表规则 1“如果 L 是最小情况”在这个问题的特定实现中是怎样转换为“如果这个链表不包含任何节点”的。这是因为当这个链表不包含任何节点时，可以合理地认定它的负数数量为 0。但是，在有些情况下，对于不包含任何节点的链表，对于问题 Q 并没有合理的答案，此时最小情况就是只包含 1 个节点的链表。假设我们的问题是：这个链表中的最大值是什么？当链表不包含任何节点时，这个问题就找不到答案。如果读者还不理解，可以试想自己是一位小学老师，并且自己所教班级的学生恰好都是女孩。如果校长问你班里有多少个男孩子是男孩合唱团的成员，你可以简单地回答零，因为你的班上没有男孩。如果校长问你班上个子最高的那个男孩叫什么名字，你就无法为这个问题提供一个合理的答案，因为班上至少要有 1 个男孩才能存在个子最高的男孩。同样，如果与一个数据集有关的问题至少需要 1 个值才能得有合理的答案，那么最小的数据集就是 1 个数据项。但是，为了保持函数的灵活性并且防止程序的崩溃，我们可能仍然想为“长度为零”的情况返回一个结果。

6.5.2 递归和二叉树

到目前为止我们所讨论的例子最多只需要一个递归调用。但是，更加复杂的数据结构可能需要多个递归调用。为了感受它的工作方式，可以考虑一种称为二叉树的数据结构。在二叉树中，每个节点包含了指向其他节点的“左”链接和“右”链接。下面是我们将要使用的类型：

```
struct treeNode {
    int data;
    treeNode * left;
    treeNode * right;
};
typedef treeNode * treePtr;
```

由于树中的每个节点指向另两个节点，因此二叉树的递归处理函数需要 2 个递归调用。我们在概念上把链表看成是 2 个部分：第 1 个节点和链表的剩余部分。为了应用递归，我们将在概念上把二叉树划分为 3 个部分：顶点的结点（称为根节点）；从根节点的左链接所能到达的所有节点（称为左子树）；从根节点的右链接所能到达的所有节点（称为右子树）。图 6.7 显示了这个概念。和处理链表一样，作为递归解决方案的开发人员，我们只要意识到左子树和右子树的存在，而不需要考虑它们的内容。图 6.8 表达了这种思路：

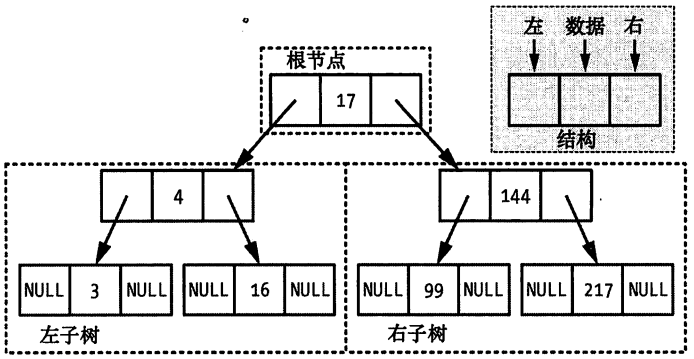

图 6.7 二叉树划分为根节点以及左右子树

和往常一样，用递归方法解决涉及二叉树的问题时，我们需要采用大递归思路。我们将执行递归函数调用，并假设它们返回正确的结果，而不需要了解递归过程是怎样解决整体问题的。和链表一样，我们对二叉树的自然划分进行操作。这就产生了下面的通用计划，以回答树 T 的问题 Q：

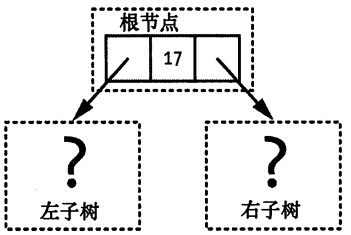

图 6.8 当程序员使用递归时应该把二叉树想像成：一个根节点，其左子树和右子树是未知的并且无需考虑的结构

1. 如果树 T 为最小规模，直接设置一个默认值。否则……
2. 执行一个递归调用，为 T 的左子树回答问题 Q。
3. 执行一个递归调用，为 T 的右子树回答问题 Q。
4. 检查 T 的根节点的值。
5. 使用上面 3 个步骤的结果，为整体的 T 回答问题 Q。

现在我们把这个通用计划应用于一个特定的问题。

寻找一棵二叉树中的最大值

编写一个函数，对一棵每个节点包含了一个整数的二叉树执行操作，返回这棵树中的最大整数。

把通用计划应用于这个特定问题产生了下面的步骤：

1. 如果树的根节点不包含任何子树，就返回根节点的值。否则……
2. 执行一个递归调用，寻找左子树的最大值。
3. 执行一个递归调用，寻找右子树的最大值。
4. 检查根节点的值。
5. 根据前面 3 个步骤返回最大的那个值。

有了这些步骤之后，我们可以直接编写解决方案的代码：

```
int maxValue(treePtr root) {
  ❶if (root == NULL) return 0;
  ❷if (root->right == NULL && root->left == NULL)
       return root->data;
  ❸int leftMax = maxValue(root->left);
  ❹int rightMax = maxValue(root->right);
  ❺int maxNum = root->data;
    if (leftMax > maxNum) maxNum = leftMax;
    if (rightMax > maxNum) maxNum = rightMax;
    return maxNum;
}
```

注意这个问题中的最小树是单节点的树❷（尽管为了安全起见覆盖了空树这种情况❶）。这是因为，我们所回答的问题只有当树中至少存在 1 个值时才有意义。考虑如果我们想把空树作为基本情况这个实际问题。这时候应该返回什么值？如果我们返回 0，就隐式地要求树中的值都为正。如果树中的所有值都是负数，那么 0 将错误地作为树中的最大值被返回。我们可以通过返回可能出现的最小（最负）整数来解决这个问题，但是这样一来当代码应用于数值类型时就必须进行小心处理了。通过把单节点作为基本情况，就可以避免这种麻烦。

代码的剩余部分非常简明。我们使用递归寻找左子树的最大值❸和右子树的最大值❹。

接着，我们使用本书中经常使用的“山丘之王”算法的一种变型寻找3个值（根节点的值、左子树的最大值和右子树的最大值）中最大的那个。

6.6 包装器函数

在本章前面的例子中，我们只讨论了递归函数本身。但是，在有些情况下，递归函数需要由另一个函数所“设置”。最常见的情况是在一个类结构的内部编写递归函数时。这可能导致递归函数所要求的参数和类的一个公共方法所需要的参数之间的不匹配。由于类一般实现了信息隐藏，因此类的客户代码可能无法访问递归函数所需要的数据或类型。下面这个例子显示了这个问题以及它的解决方案。

确定二叉树中叶节点的数量

对于一个实现了二叉树的类，添加一个公共方法，它返回树中叶节点（没有任何子树的节点）的数量。叶结点的计数应该使用递归来完成。

在实现这个问题的解决方案之前，我们首先规划这个类的概要。为了简单起见，我们只包含这个类的相关部分，忽略构造函数、析构函数甚至创建树的方法，以便把注意力集中在递归方法上。

```
class binaryTree {
    public:
      ❶int countLeaves();
    private:
        struct binaryTreeNode {
            int data;
            binaryTreeNode * left;
            binaryTreeNode * right;
        };
        typedef treeNode * treePtr;
        treePtr _root;
};
```

注意，我们的叶节点计数函数不接受任何参数❶。站在接口的角度，这是完全正确的。以下是对以前所创建的一个二叉树对象bt所进行的示例调用：

```
int numLeaves = bt.countLeaves();
```

总之，如果我们询问一棵树有多少个叶节点，对于一个对自身情况毫无所知的对象，我们能够向它提供什么信息呢？对于这个接口，情况也是这样的。用它来递归实现是完全错误的。如果没有参数，不同次的递归调用的区别在哪里呢？在这种情况下不会发生什么变化，除非通过全局变量，但这种做法正如前面所述是应该予以避免的。如果没有发生任何变化，就没有办法让递归过程继续或终止。

这个问题的解决方案是首先编写递归函数，从概念上把它当成类外部的一个函数。换句话说，我们编写这个函数对一棵二叉树的叶节点进行计数，就像我们编写函数寻找一棵二叉树的最大值一样。我们需要向这个函数传递的一个参数是一个指向节点结构的指针。

这就向我们提供了另一个使用大递归思路的机会。这种情况下的问题 Q 是什么？它是树中有多少个叶节点？把递归地处理二叉树的通用计划应用于这个特定的问题产生了下面的结果：

1. 如果树的根节点没有任何子树，这棵树总共就只有 1 个节点。按照定义，这个节点就是叶节点，因此返回 1。否则……
2. 执行一个递归调用，对左子树的叶节点进行计数。
3. 执行一个递归调用，对右子树的叶节点进行计数。
4. 在这种情况下，不需要对根节点进行检查，因为如果到达了这个步骤，根节点就不可能是叶节点。因此……
5. 返回步骤 2 和步骤 3 之和。

把这个计划转换为代码后产生了下面的结果：

```
struct binaryTreeNode {
    int data;
    treeNode * left;
    treeNode * right;
};
typedef binaryTreeNode * treePtr;
int countLeaves(treePtr rootPtr) {
    if (rootPtr == NULL) return 0;
    if (rootPtr->right == NULL && rootPtr->left == NULL)
        return 1;
    int leftCount = countLeaves(rootPtr->left);
    int rightCount = countLeaves(rootPtr->right);
    return leftCount + rightCount;
}
```

正如我们所看到的那样，这段代码是对前面的计划的直接转换。问题在于，我们怎么让这个独立的函数在这个类中使用？这是粗心大意的程序员很容易陷入麻烦的地方，我们可以考虑需要使用一个全局变量或者把根指针声明为公共部分。但是，我们不需要这样做。我们可以让一切都在类的内部发生。技巧在于使用一个包装器函数。首先，我们把这个接受 treePtr 参数的独立函数放在类的私有部分。然后，我们编写一个公共函数（也就是包装器函数），对这个私有函数进行“包装”。

由于这个公共函数可以访问私有数据成员 root，因此它可以把这个数据成员传递给此递归函数并像下面这样把结果返回给客户代码：

```
class binaryTree {
    public:
        int publicCountLeaves();
    private:
        struct binaryTreeNode {
            int data;
            binaryTreeNode * left;
            binaryTreeNode * right;
        };
        typedef binaryTreeNode * treePtr;
        treePtr _root;
        int privateCountLeaves(treePtr rootPtr);
};
❶ int binaryTree::privateCountLeaves(treePtr rootPtr) {
    if (rootPtr == NULL) return 0;
    if (rootPtr->right == NULL && rootPtr->left == NULL)
        return 1;
    int leftCount = privateCountLeaves(rootPtr->left);
    int rightCount = privateCountLeaves(rootPtr->right);
    return leftCount + rightCount;
}
❷ int binaryTree::publicCountLeaves() {
   ❸return privateCountLeaves(_root);
}
```

尽管 C++允许这两个函数具有相同的名称，但为了清晰起见我们还是使用了不同的名称以区分公共和私有的“对叶节点进行计数”的函数。PrivateCount Leaves❶的代码与前面独立的 countLeaves 函数完全相同。包装器函数 publicCountLeaves❷非常简单。它调用 privateCountLeaves，向它传递私有数据成员 root，并返回这个调用的结果❸。在本质上，它“驱动”了递归过程。包装器函数非常适用于在类的内部编写递归函数，但它们也可以在函数所需要的参数列表与调用者所传递的参数列表不匹配的任何场合使用。

6.7 什么时候选择递归

程序员新手常常疑惑为什么要采用递归。他们觉得任何程序都可以使用基本的控制结构来创建，例如选择（if 语句）和迭代（for 和 while 循环）。如果递归比基本控制结构更难以使用并且是不必要的，也许应该忽略递归。

对于这个观点有一些不同意见。首先，递归的方式编程可以帮助程序员以递归的方式进行思考，并且递归的思维方式在计算机科学的世界里是被广泛使用的，例如在编译器设计领域。其次，有些语言必须要使用递归，因为它们缺乏一些基本的控制结构。例如，纯净版的 Lisp 语句就要求在几乎所有的实质性函数中使用递归。

但是，还是有问题存在：如果一位程序员对递归进行了深入的研究并“掌握了它”，并且他使用了一种像 C++、Java 或 Python 这样的特性完整的语言，是不是还需要使用递归呢？递归在这类语言中是否有实际的用处？或者它只是一种智力上的练习？

递归的对比

为了探索这个问题，我们首先列举递归的一些不良特性。

概念复杂性

对于大多数问题，一般的程序员很难想到用递归方法来解决问题。即使理解了大递归思路，在大多数情况下使用循环仍然是更简单的解决方案。

性能

函数调用会带来显著的开销。递归调用涉及大量的函数调用，因此速度比较缓慢。

空间需求

递归并不是简单地产生许多函数调用，它还将它们嵌套在一起。也就是说，递归将形成一串长长的函数调用链，依次等待其他调用的完成。每个还没有结束的函数调用都会占用系统堆栈的空间。

乍看上去，这些特性列表给人留下的印象是递归既难又慢，而且又浪费空间。但是，这些反对意见并不是在所有情况下都成立的。在递归和迭代之间作出决定的最基本规则是：

在上面这些反对意见不成立时才选择递归。

考虑对二叉树的叶节点进行计数的函数。我们怎样在不使用递归的情况下解决这个问题呢？这是可以实现的，但是需要一种明确的机制维护节点的“面包屑尾巴”，就是那些左子树已经访问但右子树还没有被访问的节点。这些节点需要在某个时刻被重新访问，以便从右侧进行遍历。我们可以把这些节点存储在一个像堆栈这样的动态结构中。为了进行比较，下面这个函数使用了来自 C++标准模板库的 stack 类：

```
int binaryTree::stackBasedCountLeaves() {
    if (_root == NULL) return 0;
    int leafCount = 0;
  ❶stack<❷binaryTreeNode *> nodes;
  ❸nodes.push(_root);
    while (❹!nodes.empty()) {
        treePtr currentNode = ❺nodes.top();
      ❻nodes.pop();
        if (currentNode->left == NULL && currentNode->right == NULL)
            leafCount++;
        else {
            if (currentNode->right != NULL) ❸nodes.push(currentNode->right);
            if (currentNode->left != NULL) ❸nodes.push(currentNode->left);
        }
    }
    return leafCount;
}
```

这段代码所采用的模式与原来的相同，但是读者以前如果从来没有使用过 stack 类，还是需要了解一些注意事项。stack 类的工作方式就像我们在第 3 章所讨论的系统堆栈，只能在顶部添加和删除数据项。注意，我们可以使用任何没有固定长度的数据结构执行叶节点的计数操作。例如，我们也可以使用 vector。但是，使用堆栈最直接地映射了原来的代码。当我们声明这个堆栈时❶指定了将要存储在堆栈中的数据项的类型。在这个例子中，我们将存储指向 binaryTreeNode 结构的指针❷。我们在这段代码中使用了 stack 类的 4 个方法。push 方法❸把一个数据项（在这个例子中是一个节点指针）放在堆栈的顶部。empty 方法❹告诉我们堆栈中是否还有任何数据项。top 方法❺向我们提供了堆栈顶部的数据项的一份拷贝。pop 方法❻从堆栈中删除顶部那个数据项。

这个代码解决问题的思路是把指向第 1 个节点的指针放在堆栈中然后反复从堆栈上删除一个指向某个节点的指针、检查它是否为叶节点，如果是叶节点就增加计数器的值，否则就把这个节点的左孩子（如果存在）和右孩子（如果存在）放在堆栈上。因此，堆栈将追踪我们已经发现的节点，但并没有对它们进行处理，就像这个问题的递归解决方案中的

递归调用链记录了必须重新访问的节点一样。把这个迭代版本与递归版本进行比较之后，我们发现对递归的几个主要反对意见在这个例子中并不是很有说服力。首先，这段代码比递归代码更长、更复杂，因此认为递归解决方案在概念上更为复杂的反对意见在这个例子中是站不住脚的。其次，我们可以观察 stackBasedCountLeaves 创建了多少个函数调用，每访问一个内部节点（即非叶节点）时，这个函数需要进行 4 个函数调用：empty 和 top 各 1 个，还有 2 个 push。递归版本对于每个内部节点只进行两次递归调用。（注意，把堆栈的逻辑合并到函数内部从而避免对 stack 对象执行函数调用是可能的。但是，这使得函数本身的复杂性进一步增加。）最后，虽然这个迭代版本并没有使用额外的系统堆栈空间，但它明确使用了一个私有的堆栈。公平地说，它所花费的空间要比递归调用所花费的系统堆栈空间要少，但是它在系统内存上的开销仍然是与我们所遍历的二叉树的最大深度呈正比的。

在这个例子中，由于针对递归的反对意见都不存在或者被最小化了，因此递归是这个问题的好选择。从更基本的意义上说，如果一个问题用迭代方法解决起来更简单，那么迭代应该是第一选择。只有当迭代方案非常复杂时才应该使用递归。这常常涉及到我们在这里所展示的“面包屑尾巴”机制的必要性。

对树和图这样的分支结构的遍历在本质上是递归的。处理像数组和链表这样的线性数据结构则通常不需要使用递归，但是也有例外。首先用迭代方法解决问题并观察其效果肯定是不会有错的。作为本章的最后一组例子，考虑下面这些链表问题。

按顺序显示一个链表

编写一个函数，向它传递一个单链表的头指针，这个链表中每个节点的数据类型是整数。按照链表中所出现的顺序显示这些整数，每行显示一个数据。

按照相反的顺序显示一个链表

编写一个函数，向它传递一个单链表的头指针，这个链表中每个节点的数据类型是整数。按照与链表中的出现顺序相反的顺序显示这些整数。

由于这两个问题是同一个问题相反的两端，因此很自然会想到它们的实现也将是同一个实现的相反两端。对于递归实现，事实上也确实如此。使用前面所给出的 listNode 和

listPtr类型，下面是解决这两个问题的递归函数：

```
void displayListForwardsRecursion(listPtr head) {
    if (head != NULL) {
        ❶cout << head->data << "\n";
        ❷displayListForwardsRecursion(head->next);
    }
}
void displayListBackwardsRecursion(listPtr head) {
    if (head != NULL) {
        ❸displayListBackwardsRecursion(head->next);
        ❹cout << head->data << "\n";
    }
}
```

正如我们所看到的那样，这两个函数的代码基本上相同，区别只在于 if 语句内部的两条语句的顺序。所有的区别都在这个地方。在第一种情况下，我们在执行递归调用显示链表剩余部分❷之前先显示第 1 个节点的值❶。在第二种情况下，我们先执行递归调用显示链表的剩余部分❸，然后再显示第 1 个节点的值❹。这就产生了正好相反的显示顺序。

由于这两个函数都非常简明，因此我们可能觉得递归用于解决这两个问题是非常合适的。但是，事实并非如此。为此，我们观察这两个函数的迭代实现。

```
void displayListForwardsIterative(listPtr head) {
   ❶for (listPtr current = head; current != NULL; current = current->next)
        cout << current->data << "\n";
}
}
void displayListBackwardsIterative(listPtr head) {
   ❷stack<listPtr> nodes;
   ❸for (listPtr current = head; current != NULL; current = current->next)
        nodes.push(current);
   ❹while (!nodes.empty()) {
      ❺nodePtr current = nodes.top();
      ❻nodes.pop();
      ❼cout << current->data << "\n";
    }
}
```

按原来的顺序显示链表的函数非常简单，就是一个直接的遍历循环❶，正如我们在第 4 章所看到的那样。但是，以相反顺序显示链表的函数就复杂得多了。它遇到了与二叉树问题的“面包屑尾巴”相同的要求。按照定义，以反序显示一个链表的节点要求返回到前面的节点。在单链表中，通过链表本身是无法做到这一点的，因此需要使用另一个结构。在这个例子中，我们需要另一个堆栈。在声明了这个堆栈❷之后，我们使用一个循环把链表中

的所有节点压入这个堆栈中❸。由于这个是堆栈，每个被添加的数据项都被压入到以前的数据项的顶部，因此这个链表的第 1 个数据项将出现在堆栈的底部，而链表的最后一个数据项将出现在堆栈的顶部。我们执行一个终止条件为堆栈为空的 while 循环❹，反复提取指向堆栈顶部的那个节点的指针❺，从堆栈删除这个节点指针❻，并显示所引用的节点的数据❼。由于堆栈顶部的数据是链表的最后一个数据，因此它相当于以相反的顺序显示链表的数据。

与前面所显示的对二叉树进行迭代的函数一样，在编写这个函数时也可以不使用堆栈（在函数中创建另一个链表，与原来链表的顺序相反）。但是，没有办法使第 2 个函数像第 1 个函数那样简单，也没有办法从遍历 2 个结构削减为只遍历 1 个。对递归和迭代实现进行比较之后，很容易发现迭代的“前向”函数极为简单，因此采用递归实现没有任何优势，只会存在一些缺点。相反，递归的“反向”函数比迭代版本要简单得多，其执行效率预计也和迭代版本差不多。因此，“反向”函数采用递归方法是合理的，而“前向”函数虽然可以作为一个练习递归的好例子，但在实际使用中并没有理由选择递归实现。

6.8 习题

和往常一样，对本章所讨论的思路进行实践是极为重要的。

6.1 编写一个函数，计算一个整数数组中所有正数的和。首先，使用迭代解决这个问题。接着，使用本章所讨论的技巧，把迭代函数转换为递归函数。

6.2 考虑一个表示二进制字符串的数组，其中每个元素的数据是 0 或者 1。编写一个 `bool` 函数，确定这个二进制字符串是否具有奇校验（1 的位数是否为奇数）。提示：递归函数应该返回 true（表示奇）或 false（表示偶），而不是 1 的位数。首先用迭代解决这个问题，然后用递归解决这个问题。

6.3 编写一个函数，它接受一个整数数组以及一个“目标”数为参数，返回这个目标数在这个数组中的出现次数。首先用迭代解决这个问题，然后改用递归解决。

6.4 自行设计：找到一个自己已经解决或者按照自己当前的技术水平解决起来完全没有问题的一维数组的问题，用递归的方式解决这个问题。

6.5 再次解决习题 6.1，使用链表代替数组。

6.6 再次解决习题 6.2，使用链表代替数组。

6.7 再次解决习题 6.3，使用链表代替数组。

6.8 自行设计：尝试找到一个很难用迭代解决但可以用递归直接解决的链表处理问题。

6.9 编程中的有些术语具有不止一个含义。在第 4 章中，我们学习了堆的概念，也就是可以用 new 分配内存的地方。堆这个术语还可以用于描述二叉树中每个节点的值比它的左子树或右子树中的任何节点的值更大的情况。编写一个递归函数，确定一棵二叉树是否是一个堆。

6.10 二叉搜索树是指一棵二叉树的每个节点的值都大于它的左子树的所有节点的值但小于它的右子树的所有节点的值。编写一个递归函数，确定一棵二叉树是否为二叉搜索树。

6.11 编写一个递归函数，接受一个指向一棵二叉搜索树的根节点的指针和一个需要插入的新值为参数，用这个新值创建一个新节点，把它放在这棵二叉树中正确的位置，继续保持二叉搜索树的结构。提示：考虑把根节点指针参数声明为引用参数。

6.12 自行设计：思考与一组数值有关的基本统计问题，例如平均值、中值、众数等。尝试编写一个递归函数，为一棵包含整数的二叉树计算这些统计数据。有些统计数据要比另一些容易统计得多，为什么？

第 7 章

通过代码复用解决问题

本章与本书之前的章节存在很大的区别。在前面的章节中，我强调了亲自寻找问题的解决方案的重要性。不管怎样，这正是本书的目标：为编程问题编写初步的解决方案。但是，在前面的章节中，我们也谈到了总是可以从以前所编写的代码中受益，这也是为什么应该保留自己所编写的所有代码供未来参考的原因。在本章中，我们将在这方面深入一步，讨论怎样复用其他程序员的代码和思路来解决我们的问题。

如果读者还记得本书是怎么开始的，那么本章的话题可能显得有些突兀。在一开始，我谈到了通过修改其他人的代码来解决复杂问题是种错误的做法。这种做法不仅成功的机率很低，而且就算成功也不会向我们提供什么学习经验。如果读者以前都是按照这种方式进行编程，那么就无法成长为一名真正的程序员，在软件开发领域前景也是非常有限。我想说的是，一旦任何问题达到了一定程度的规模，期望程序员从头开发一个解决方案是不太现实的。这会导致程序员把大量的时间浪费在低效率的工作中，并且极大地依赖程序员必须精通各个方面的知识。另外，这种做法很容易导致程序充满缺陷或难以维护。

7.1 良好的复用和不良的复用

因此，我们必须区分良好的复用和不良的复用。良好的复用帮助我们编写更好的程序并且提高程序的编写速度。不良的复用可能短时间内帮助我们借用其他程序员的思维，但最终会导致不良的开发，不论对于代码还是程序员都是如此。表 7.1 对它们之间的区别进行了总结。左边一列显示了良好复用的属性，右边一列显示了不良复用的属性。在考虑是否对代码进行复用时，要考虑它很可能会产生左边一列的属性还是右边一列的属性。

表 7.1 良好的复用和不良的复用

良好的复用	不良的复用
遵循某个蓝图	复制其他人的工作
强化并扩展自己的能力	伪造自己的能力
帮助自己学习	无法帮助自己学习
不论从短期还是长期都可以节省时间	可能在短期节省时间，但从长期来看会延长时间
产生满足要求的程序	可能产生无法满足要求的程序

值得注意的是，良好的复用和不良的复用之间的区别并不是我们复用了什么代码或者我们怎样复习它们，而是在子我们所借用的代码和概念之间的关系。以前在文学课上撰写一篇学期论文时，我发现在以前的课程中所学习的东西与我的论文的主题是相关的，因此我就在论文中吸收了它们。当我把论文的草稿提交给教授时，她告诉我必须注明这些信息的引用。深感挫折之后，我询问教授可以在论文的什么地方简单地陈述自己的知识而不需要提供引用。她的答案是当我成为专家，其他人都在引用我，当所有一切都在我的头脑中时，我就可以不再引用其他人了。

按照编程的术语，良好的复用是指我们通过阅读某个人对一个基本概念的描述后编写代码或者利用自己以前所编写的代码。在本章中，我们将讨论如何获得编码概念的所有权，以保证我们所进行的复用可以帮助自己成为更好的程序员，而不仅仅是为了偷懒。

我们把注意力转向表 7.1 的最后一行。不良的复用常常会导致失败。这并不令人惊奇，因为有可能是一位程序员在复用自己实际上并不理解的代码。在有些情况下，被借用的代码一开始能够工作，但是当程序员试图对借用的代码进行修改或者扩展时，由于缺乏深入的理解，很难用有组织的方式完成这样的任务。程序员必须通过不断的尝试和失败来进行试验，这样就违背了我们的基本问题解决规则的第一条也是最重要一条：总是要制订计划。

7.2 组件基础知识回顾

知道了我们打算采用的复用类型之后，现在可以对代码的不同复用方法进行分类了。在本书中，我打算用组件这个术语来表示由一位程序员所创建的、可以被其他人复用以帮助解决编程问题的任何东西。组件可以在任何地方出现，它可以是抽象的，也可以是具体的。它可以是一个思路，也可以是一段完整实现的代码。如果我们把解决一个编程问题看成是处理一个手工项目，我们所学会的解决问题的技巧就像工具一样，而组件就像是专用的零件。下面的每种组件都是复用程序员的以前工作的不同方式。

代码块

代码块就是将一块代码从一个程序清单复制到另一个程序清单。按照更通俗的说法，我们可以称为复制粘贴工作。这是最低级形式的组件用法，常常代表了不良的复用，会出现不良复用可能导致的所有问题。当然，如果被复制的代码是自己所编写的，那就不会有实际的危害，但最好还是把一段现有的代码包装成为一个类库或其他结构，允许它以一种更清晰和更容易维护的方式被复用。

算法

算法是一种编程配方，它是完成一个目标的特定方法，是以日常语言或流程图的形式表达的。例如，在第 3 章中，我们讨论了数组的排序操作，并讨论了排序的不同实现方式。对数组进行排序的一种方式是插入排序算法，并且我演示了这种算法的一个示例实现。值得注意的是，这段特定的代码只是插入排序的一种实现，而插入排序是指算法本身，也就是对数组进行排序的方法，而不是指特定的代码。插入排序的工作原理是反复地取数组中下一个未排序的值，并把已排序的值“挪动”一个位置，直到找到了我们当前所插入的值在数组中的正确位置。使用这种方法对数组进行排序的任何代码都可以作为插入排序的实现。

算法是一种高级形式的复用，一般具有良好的复用属性。算法在本质上只是思路。作为程序员，必须根据自己的编程技能以及对算法本身的深入理解来实现这种思路。我们将要使用的算法通常是经过了仔细的研究，在各种情况下具有可预测的表现。有了算法作为蓝图之后，我们就对代码的正确性和性能有了充分的信心。

但是，把代码建立在算法的基础上还是存在一些潜在的弱点。当我们使用一个算法时，是从一个概念层次开始工作的。因此，在到达实现那部分程序的最终代码之前还有很长的

路要走。算法确实能节省时间，因为它从本质上来说在解决问题方面是完整的，但是取决于算法本身以及它在编程中的特定应用，算法的实现可能并不简单。

模式

在编程中，模式（或设计模式）表示具有一种特定编程技巧的模板。这个概念与算法有关，但又存在区别。算法就像解决特定问题的配方，而模式是在特定的编程情况下所使用的基本技巧。模式所解决的问题一般是在代码本身的结构内部。例如，我们在第 6 章讨论了在链表类中由一个递归函数所表示的问题：递归函数需要指向链表第 1 个节点的“头”指针作为它的参数，但数据要求保持私有。解决方案是创建一个包装器函数，它把一个参数列表适配到另一个。包装器技巧就是一种设计模式。我们可以使用这种模式解决一个类中的递归函数问题，但它也可以用于其他途径。例如，我们有一个 `linkedList` 类，它允许在链表的任何位置插入或删除数据项，但是自己所需要的是一个堆栈类，也就是只允许在一端进行插入和删除的列表。我们可以创建一个新的堆栈类，实现一些公共方法以完成典型的堆栈操作，例如压入和弹出。这些方法将只是调用作为这个堆栈类的私有数据成员的一个 `linkedList` 对象的成员函数。通过这种方式，我们可以复用链表类的功能，同时提供堆栈类的接口。

和算法一样，模式也是高级形式的组件复用，学习模式也是创建自己的编程工具箱的非常好的方法。但是，模式也存在与算法相同的一些问题。知道了一个模式的存在并不等于知道了怎样用我们为一个编程解决方案所选择的特定语言实现这个模式，并且要想正确地实现模式及最优性能往往是比较复杂的。例如，有一个称为 singleton 的模式，符合这个模式的类只允许创建这个类的 1 个对象。创建一个 singleton 类非常简单，但创建只在实际需要时才能创建一个实例对象的 singleton 类却非常困难，而最好的解决技巧可能因不同的语言而异。

抽象数据类型

正如我们在第 5 章所讨论的那样，抽象数据类型是由它的操作而不是由这些操作的实现方法所定义的类型。我们在本书中多次使用的堆栈类型就是一个很好的例子。抽象数据类型与模式的相似之处在于它们定义了操作的效果，但并没有特别地定义这些操作的实现方式。但是，和算法一样，这些操作存在一些众所周知的实现技巧。例如，堆栈可以用任意数据的底层数据结构来实现，例如链表或数组。但是，一旦我们决定了使用一种特定的数据结构，有时候就已经决定了用什么方式实现。假设我们用链表实现一个堆栈，但是无法对一个现有的链表进行包装，而是必须自己编写链表代码。由于堆栈是一种后入先出的

结构，它只有在链表的一端进行插入和删除才是合理的。而且，它只有在链表的头部进行插入和删除才是合理的。从理论上说，我们也可以在尾部进行插入和删除，但这种做法会在每次插入和删除时产生低效的链表遍历。为了避免这种遍历，就需要一种双链表，由一个单独的指针指向链表的最后一个节点。在链表的头部进行插入和删除是最简单和最有效的实现，因此堆栈的链表实现几乎都采用了相同的方式。

因此，即使抽象数据类型中的抽象这个词意味着类型是概念性的并且不包含实现细节，但是在实践中，当我们选择在代码中实现一种抽象数据类型时，我们并不会从头创建这种实现，而是用该类型的一种现有实现作为指导。

库

在编程中，库表示一些相关代码片段的集合。库一般包含了已编译形式的代码以及所需的源代码声明。库可以包含独立的函数、类、类型声明以及任何可以出现在代码中的东西。在 C++中，最显而易见的例子就是标准库。我们在前面章节中所使用的 `strcmp` 函数就来自旧式的 C 函数库 cstring，像 vector 这样的容器类则来自 C++的标准模板库，乃至像我们在基于指针的代码中所使用的 `NULL` 也不是 C++语言本身固有的东西，而是在一个库头文件 stdlib.h 中所定义的。由于有太多的核心功能都是包含在库中的，因此在现代的编程中，库的使用是不可避免的。

一般而言，库的使用是良好的代码复用。代码被包含在库中，因为它提供了各种程序一般需要使用的功能。库的代码可以帮助程序避免“重新发明轮子”。然而，作为程序开发人员，当我们使用库代码时，必须从中学到些什么，而不是单纯走捷径。在本章的后面，我们将看到关于这方面的一个例子。

注意，虽然有许多库是以通用为目的的，但也有一些被设计为应用程序编程接口（API）的形式，向高级编程语言提供底层平台的简化视图或更连贯的视图。例如，Java 语言包含了一个称为 JDBC 的 API，它提供了一种标准的方式，允许程序与关系数据库进行交互。另一个例子是 DirectX，它向 Microsoft Windows 游戏开发人员提供了声音和图像方面的广泛功能。在这两个例子中，库提供了高级程序和基础层次的硬件和软件（在 JDBC 中是数据库引擎，在 DirectX 中是图像和声音硬件）之间的连接。而且，在这两个例子中，代码复用不仅仅是良好的，而且从任何实用的观点来看都是必需的。Java 的数据库程序员或者为 Windows 编写 C++代码的图形程序员肯定需要使用这些 API。如果没有这些 API，也必须使用其他相似的东西，程序员不可能从头开始创建与平台的一个新连接。

7.3 创建组件的基础知识

组件非常有用，因此程序员应该尽可能地利用组件。但是，为了利用一个组件来帮助解决一个问题，程序员必须知道它的存在。取决于对组件定义的精细程度，可用的组件可能高达数百个甚至数千个，而程序员新手所看到的组件往往只有其中几个。因此，优秀的程序员必须养成习惯向他的工具箱中不断添加组件知识。对组件知识的收集可以通过两种不同的方式进行：程序员可以明确分配时间学习新组件，把它作为一项基本任务，或者可以搜索一个组件来解决一个特定的问题。我们把第一种方法称为探索式学习，把第二种方法称为根据需要学习。为了提高自己的水平，这两种学习方式都是必要的。一旦精通了所选择的编程语言的语法之后，发现新组件就是完善程序员技能的主要方法之一。

7.3.1 探索式学习

我们首先从一个探索式学习的例子开始。假设我们想学习关于设计模式的更多知识。幸运的是，对于最实用和最频繁使用的设计模式，人们已经达成了非常广泛的一致。因此，关于这个话题，我们能够接触大量的资源，可以相当地确信没有遗漏重要的东西了。通过简单地查找一些设计模式并对它们进行研究，就可以从中受益。如果我们实现了其中一些模式，就可以得到更多的收获。

我们在典型的模式列表中可以找到的一种模式叫策略。这是一种思路，允许一种算法（或算法的一部分）在运行时才被选择。在策略模式的最基本形式中，它允许更改函数或方法的操作方式，但不允许对结果进行更改。例如，一个对类的数据进行排序（或者参与了排序过程）的类方法可能允许对排序的方法做出选择（如选择快速排序或插入排序）。不管选择什么排序方式，其结果（排序后的数据）是相同的，但是允许客户选择排序方法可能使代码的执行效率更高。例如，客户对于具有很高重复率的数据可以避免选择快速排序。根据策略的形式，客户的选择会对结果产生影响。例如，有一个表示一手牌的类，排序策略可能会决定 A 被认为是最大（比 K 大）还是最小（比 2 小）。

把学习融入到实践中

阅读了上面这段文字之后，读者现在知道了什么是策略模式，但还没有运用于创建自

己的实现。在五金商店浏览工具和实际购买并使用工具之间还是存在显著区别的。因此，我们现在从货架上取出这个设计模式，并把它投入到使用中。尝试新技巧的最快方法是把它融入到已经编写完成的代码中。让我们设计一个可以用这种模式解决的问题，它建立在我们已经编写完成的代码基础之上。

班长

在一所特定的学校中，每个班级具有一名指定的“班长”，如果教师离开了教室，就由这名学生负责维持课堂秩序。最初，这个称号授予班里学习成绩最好的学生。但是，现在有些教师觉得班长应该是具有最深资历的学生，也就是学生 ID 最小的那个学生，因为学生 ID 是按照进入班级的先后顺序依次分配的。还有一部分教师觉得指定班长这个传统是件非常愚蠢的事情。为了表示抗议，他们简单地选择按照字母顺序排列的班级花名册中所出现的第 1 个学生。我们的任务是修改学生集合类，添加一个方法从集合中提取班长，同时又能适应不同教师的选择标准。

正如我们所看到的那样，这个问题将要采用策略形式的模式。我们需要让这个方法根据不同的选择标准返回不同的班长。为了在 C++中实现这一点，需要使用函数指针。我们已经简单地在第 3 章的 qsort 函数中了解过这个概念。qsort 函数接受一个函数指针，它所指向的函数对需要进行排序的数组中的两个数据项进行比较。在这个例子中，我们完成类似的任务。我们将创建一组比较函数，接受 2 个 studentRecord 对象为参数并分别根据成绩、学生 ID 值或姓名确定第 1 个学生是否“好于”第 2 个学生。

首先，我们需要为比较函数定义一个类型：

```
typedef bool ❶(* firstStudentPolicy)(studentRecord r1, studentRecord r2);
```

这个声明创建了一个称为 firstStudentPolicy 的类型，它是个函数指针，它所指向的函数返回一个 bool 值并接受两个 studentRecord 类型的参数。*firstStudentPolicy 号两边的括号❶是必要的，这是为了防止这个声明被解释为返回一个 BOOL 类型的指针的函数。有了这个声明之后，我们就可以创建 3 个策略函数了：

```
bool higherGrade(studentRecord r1, studentRecord r2) {
   return r1.grade() > r2.grade();
}
bool lowerStudentNumber(studentRecord r1, studentRecord r2) {
   return r1.studentID() < r2.studentID();
}
bool nameComesFirst(studentRecord r1, studentRecord r2) {
   return ❶strcmp(r1.name().c_str()❷, r2.name().c_str()❷) ❸< 0;
}
```

前两个函数非常简单：higherGrade 在第 1 条记录的成绩值大于第 2 条记录时返回 true，lowerStudentNumber 在第 1 条记录的学生 ID 值小于第 2 条记录时返回 true。第 3 个函数 nameComesFirst 在本质上与前两个函数相同，但它需要使用 strcmp❶库函数。这个函数接受 2 个“C 风格”的字符串，即以 null 结尾的字符数组而不是 string 对象。因此我们必须对两条学生记录的姓名字符串使用 c_str()方法❷。strcmp 函数在第 1 个字符串按照字母顺序出现在第 2 个字符串之前时返回一个负数，因此我们检查它的返回值以判断它是否小于 0❸。现在，我们就可以修改 studentCollection 类本身了：

```
class studentCollection {
private:
   struct studentNode {
      studentRecord studentData;
      studentNode * next;
   };
public:
   studentCollection();
   ~studentCollection();
   studentCollection(const studentCollection &copy);
   studentCollection& operator=(const studentCollection &rhs);
   void addRecord(studentRecord newStudent);
   studentRecord recordWithNumber(int IDnum);
   void removeRecord(int IDnum);
 ❶void setFirstStudentPolicy(firstStudentPolicy f);
 ❷studentRecord firstStudent();
private:
 ❸firstStudentPolicy _currentPolicy;
   typedef studentNode * studentList;
   studentList _listHead;
   void deleteList(studentList &listPtr);
   studentList copiedList(const studentList copy);
};
```

这个类声明在第5章的基础上增加了3个新的成员：一个私有数据成员_currentPolicy❸，它存储了指向其中一个策略函数的指针、一个用于修改策略的 setFirstStudentPolicy 方法❶以及根据当前策略返回班长的 firstStudent 方法本身❷。setFirstStudentPolicy

的代码非常简单：

```
void studentCollection::setFirstStudentPolicy(firstStudentPolicy f) {
   _currentPolicy = f;
}
```

我们还需要修改默认构造函数对当前策略进行初始化：

```
studentCollection::studentCollection() {
   _listHead = NULL;
   _currentPolicy = NULL;
}
```

现在，我们可以编写 `firstStudent` 方法：

```
studentRecord studentCollection::firstStudent() {
 ❶if (_listHead == NULL || _currentPolicy == NULL) {
      studentRecord dummyRecord(-1, -1, "");
      return dummyRecord;
   }
   studentNode * loopPtr = _listHead;
 ❷studentRecord first = loopPtr->studentData;
 ❸loopPtr = loopPtr->next;
   while (loopPtr != NULL) {
      if (❹_currentPolicy(loopPtr->studentData, first)) {
         first = loopPtr->studentData;
      }
    ❺loopPtr = loopPtr->next;
   }
   return first;
}
```

这个方法首先检查特殊情况。如果没有需要检查的链表或者不存在策略❶，就返回一条哑记录。否则，就使用本书中广泛使用的基本搜索技巧，对这个链表进行遍历并寻找最适当地匹配当前策略的学生。我们把链表开始位置的那条记录赋值给 `first`❷，使循环变量从链表的第 2 条记录开始❸，然后执行遍历。在遍历循环中，对当前策略函数的调用❹告诉我们目前所查看的学生根据当前标准是否“好于”到现在为止所找到的最佳学生。当这个循环结束时，我们就返回“班长”❺。

班长解决方案的分析

使用策略模式解决了一个问题之后，我们很可能想确认这种技巧可以适用的其他场合，而不是一次了解了这个技巧之就将其束之高阁。我们还可以对示例问题进行分析，形成对这个技巧的价值的认识，明白什么时候使用它比较合适，什么时候使用它则是个错误，或

至少它带来的价值应多于麻烦。对于这个特定的模式，读者可能会看到它弱化了封装和信息隐藏。例如，如果客户代码提供了策略函数，它就需要访问通常属于类内部的类型，在这个例子中也就是 studentRecord 类型。（我们将在习题中思考一种处理这个问题的方法。）这意味着如果我们修改了这个类型，客户代码就有可能失败。把这个模式应用于其他项目之前，必须在这个顾虑与它可能带来的好处之间进行权衡。在前面的章节中，我们讨论了怎样理解什么时候应该使用一种技巧以及在什么时候不应该使用这种技巧，这个问题的重要性并不亚于理解怎样使用这个技巧。通过对自己的代码进行检查，可以对这个关键问题获得深入的体会。

至于进一步的实践，我们可以检查已完成项目的库，搜索可以使用这种技巧进行重构的代码。记住，很多“现实世界”的编程涉及到对现有的代码进行补充或修改，因此这是进行这类修改的一种非常好的实践，还能发展自己运用某种特定组件的技能。而且，良好的代码复用的一个优点是我们可以从中进行学习，而实践能够最大限度地提升学习的效果。

7.3.2 根据需要学习

前一节描述了“漫游式学习”的过程。虽然这种类型的学习旅程对于程序员而言是极具价值的，但有时候我们必须直接针对一个特定的目标学习。如果我们正在着手处理一个特定的问题，特别是当这项工作面临极大的时间压力时，我们会猜测某个组件可能会为我们提供极大的帮助。我们不想通过随机漫游编程世界来碰到自己所需要的东西，而是想尽可能快地找到直接适用于自己所面临问题的组件。但是，这听起来似乎有些怪异，当我们并不准确地知道自己所寻找的是什么时，怎样才能找到自己所需要的东西呢？思考下面这个示例问题：

高效的遍历

一个编程项目将使用我们的 studentCollection 类。客户代码需要做到能够遍历集合中的所有学生。显然，为了维护信息隐藏，客户代码不能直接访问这个链表，但要求高效地对其进行遍历。

由于这段描述中的关键词是高效，让我们精确地分析它在这个例子中的含义。我们假设 studentCollection 类的一个特定对象具有 100 条学生记录。如果我们直接访问这个链表，可以编写一个迭代 100 次的循环。这是所有的链表遍历中最高效的做法。任何要求

我们迭代超过 100 次的循环都可以认为其结果是不够高效的。

如果没有高效这个需求，我们可以在这个类中添加一个简单的 recordAt 方法来解决这个问题。这个方法返回集合中特定位置的学生记录，第 1 条记录的位置编号为 1：

```
studentRecord studentCollection::recordAt(int position) {
   studentNode * loopPtr = _listHead;
   int i = 1;
 ❶while (loopPtr != NULL && i < position) {
      i++;
      loopPtr = loopPtr->next;
   }
   if (loopPtr == NULL) {
    ❷studentRecord dummyRecord(-1, -1, "");
      return dummyRecord;
   } else {
    ❸return loopPtr->studentData;
   }
}
```

在这个方法中，我们使用了一个循环❶对链表进行遍历，直到找到了所需的位置或者到达了链表的尾部。当这个循环结束时，如果已经到达了链表的尾部，我们就创建并返回一条哑记录❷。如果是在指定的位置就返回这条记录❸。问题在于我们执行遍历只是为了寻找一条学生记录。这并不一定是完整的遍历，因为当我们到达所需的位置时就会终止循环，但它终归还是进行了遍历。假设客户代码试图求学生成绩的平均值：

```
int gradeTotal = 0;
for (int recNum = 1; recNum <= numRecords; recNum++) {
   studentRecord temp = sc.recordAt(recNum);
   gradeTotal += temp.grade();
}
double average = (double) gradeTotal / numRecords;
```

对于这段代码，假设 sc 是个以前所声明并生成的 `studentCollection` 对象，`recNum` 是个表示记录数量的整数。假设 `recNum` 变量值为 100。当我们初步扫视这段代码时，可能觉得计算平均成绩只需要迭代这个循环 100 次，但由于每次调用 `recordAt` 函数本身就要执行一次不完整遍历，因此这段代码总共涉及 100 次遍历，每次遍历平均需要进行 50 次迭代。因此，它的结果并不是非常高效的 100 个步骤，而是大约需要 5000 个步骤，这是极为低效的。

什么时候搜索组件

现在，我们触及到真正的问题。让客户访问集合成员对其进行遍历是非常容易的，但

高效地提供这种访问却是非常困难的。当然，我们可以尝试只用自己的能力来解决这个问题。但是，如果我们可以使用一个组件，就能够很快实现一个解决方案。为了寻找一个适用于我们的解决方案的未知组件，第 1 个步骤是假设这个组件实际上存在。换句话说，如果我们不开始搜索，就肯定无法找到这样一个组件。因此，为了最大限度地获得组件的优点，需要使自己处于能够让组件发挥作用的场合。发现自己陷在问题的某个方面而无法自拔时，可以尝试下面这些方法：

1. 以通用的方式重新陈述这个问题。

2. 向自己提问：这是否可能成为一个常见的问题？

第 1 个步骤非常重要，因为我们把问题陈述为“允许客户代码高效地计算一个类所封装的记录链表中的平均学生成绩”，它听上去特定于我们所面临的情形。但是，如果我们把这个问题陈述为“允许客户代码高效地遍历一个链表，并且不需要提供对链表指针的直接访问”，我们就开始理解这可能成为一个常见的问题。显然，我们可以想象，由于程序常常需要在类中存储链表和其他线性访问的数据结构，因此其他程序员肯定已经想出了允许高效地访问数据结构中的每个数据项的办法。

寻找组件

既然我们已经同意进行观察，现在就可以寻找组件了。为了清晰起见，我们把原来的编程问题重新陈述为一个搜索问题：“寻找一个组件，可以用它修改我们的 `studentCollection` 类，允许客户代码高效地遍历内部的链表。”那么怎样解决这个问题呢？我们首先可以观察任意类型的组件：模型、算法、抽象数据类型或库。

假设我们首先在标准 C++库中进行寻找。我们没有必要寻找一个可以“插入”到自己的解决方案中的类，而是挖掘一个与自己的 `studentCollection` 类相似的库类，以借鉴思路。这就用到了我们用于解决编程问题的类比策略。以前对 C++库的探索已经使我们与诸如 `vector` 这样的容器类有了一定程度的接触，因此我们应该寻找一种与学生集合类最为相似的容器类。如果求助于自己所喜欢的 C++参考资料，例如一本相关的书籍或网络上的一个站点并查看 C++容器类，将会发现有一个称为 list 的“线性容器”符合这个要求。list 类是否允许客户代码对它进行高效的遍历呢？它能够做到这一点，只要使用一个称为迭代器的对象。我们看到 `list` 类提供了产生迭代器的 `begin` 和 `end` 方法。迭代器是一种对象，它可以引用 `list` 类中的一个特定数据项，并且可以增加自己的值，使它引用 `list` 类中的下一个对象。如果 `integerList` 是一个包含了整数的 `list<int>`并且 `iter` 是个 `list<int>::iterator`，我们就可以用下面的代码显示这个 `list` 中的所有整数：

```
iter = intList.begin();
while (iter != intList.end()) {
   cout << *iter << "\n";
   iter++;
}
```

通过使用迭代器，list 类向客户代码提供了一种机制高效地对 list 进行遍历，从而解决了这个问题。此时，我们可能会想到把 list 类本身吸收到我们的 studentCollection 类中，替换原先所使用的链表。然后，我们可以为这个类创建 begin 和 end 方法，对它所包含的 list 对象的方法进行包装，这样问题就解决了。但是，这种做法就涉及到良好的复用和不良的复用的问题。一旦我们完全理解了迭代器的概念并且可以在自己的代码中生成它，再把标准模板库中的一个现有的类插入到自己的代码中就是非常好的选择，甚至是最好的选择。但是，如果我们没有能力做到这一点，对 list 类的这种偷懒用法就不会帮助自己成长为优秀的程序员。当然，有时候我们必须使用那些自己无法生成的组件，但是如果我们养成了让其他程序员为自己解决问题的习惯，就很难成长为真正的问题解决专家。

因此，让我们自己实现迭代器。在此之前，我们先简单地观察一下寻找迭代器方法的其他途径。我们是在标准模板库中进行搜索的，但也可以从其他地方开始搜索。例如，我们也可以在一组常用的设计模式中进行搜索。在“行为模式”这个标题的下面，我们可以找到迭代器模式。在这个模式中，客户允许对集合中的数据项进行线性访问，而不需要直接接触集合的底层结构。这正是我们所需要的。我们可以通过搜索一个模式列表找到它，也可以通过以前对模式的研究想到它。我们还可以从抽象数据类型开始搜索，因为通用意义上的列表（以及特定意义上的链表）是常见的抽象数据类型。但是，对列表抽象数据类型的许多讨论和实现并没有考虑到把客户对列表的遍历作为一种基本操作，因此不会引发迭代器的概念。最后，如果我们是在算法领域开始搜索的，很可能无法找到适用的东西。算法倾向于描述技巧性的代码，而创建迭代器则相当简单，正如我们稍后将要看到的那样。在这个例子中，在类库中搜索使我们以最快的速度找到了目标，其次是模式。但是，作为一个基本规则，在搜索一个有用的组件时，必须考虑所有的组件类型。

应用组件

现在，我们准备为 studentCollection 类创建一个迭代器，但是标准模板库的 list 类向我们所展示的只是怎样在客户代码中使用迭代器的方法。如果我们不知道该怎样实现迭代器，可以考虑对 list 类以及它的祖先类的代码进行研究，但是阅读大量不熟悉的代码

无疑具有很大的难度，是万般无奈的情况下不得已采用的办法。其实，我们可以用自己的方式来对它进行思考。把以前的代码例子作为参考，我们可以认为迭代器是由 4 个核心操作所定义的：

1. 集合类提供了一个方法 ，提供了引用集合第 1 个元素的迭代器。在 list 类中，这个方法是 begin。
2. 测试迭代器是否越过了集合最后一个元素的机制。在 list 类中，这个方法是 end，它针对这个测试产生了一个特殊的迭代器对象。
3. 迭代器类中使迭代器向前推进一步的方法，是使它引用集合中的下一个元素。在 list 类中，这个方法是重载的++操作符。
4. 迭代器类中返回集合当前所引用的元素的方法。在 list 类中，这个方法是重载的*（前缀形式）操作符。

站在编写代码的角度，上面这些并没有困难之处，唯一的问题就是把所有的东西放在正确的位置。因此，我们现在就开始处理这个问题。根据上面的描述，我们的迭代器（称为 scIterator）需要存储一个指向 studentCollection 中的一个元素的引用，并且能够推进到下一个元素。因此，这个迭代器应该存储一个指向 studentNode 的指针，这样就允许它返回集合中的 studentRecord 对象并允许它推进到下一个 studentNode 对象。因此，这个迭代器类的私有部分将具备以下这个数据成员：

```
studentCollection::studentNode * current;
```

我们马上就遇到了一个问题。studentNode 类型是在 studentCollection 类的私有部分声明的，因此上面这行代码是行不通的。我们首先想到的是不应该把 studentNode 声明为私有部分，但这并不是正确的答案。节点类型在本质上是私有的，因为我们并不希望任何客户代码依赖节点类型的某种特定性质实现，不想因为这个类进行了修改而导致客户代码的失败。然而，我们还是需要让 scIterator 类能够访问自己的私有类型。我们通过一个友元声明来解决这个问题。在 studentCollection 类的公共部分，我们添加了下面这一行：

```
friend class scIterator;
```

现在，scIterator 可以访问 studentCollection 类的私有声明，包括 studentNode 的声明。我们还可以声明一些构造函数，如下所示：

```
scIterator::scIterator() {
   current = NULL;
}
scIterator::scIterator(studentCollection::studentNode * initial) {
   current = initial;
}
```

我们稍微观察一下 studentCollection 类再编写 begin 方法，这个方法返回一个引用集合第 1 个元素的迭代器。根据本书所使用的命名方案，这个方法应该用名词来表示，例如 firstItemIterator：

```
scIterator studentCollection::firstItemIterator() {
   return scIterator(_listHead);
}
```

正如所看到的那样，我们需要完成的任务就是把链表的头指针塞到一个 scIterator 对象中并返回它。如果读者的做事风格与我相似，看到指针飞临此处可能会觉得有点紧张，但是注意 scIterator 正要保存一个指向 studentCollection 列表中的一个元素的引用。它不会为自己分配任何内存，因此我们并不需要担心深拷贝和重载的赋值操作符。

现在我们返回到 scIterator 并编写其他方法。我们需要一个方法推进迭代器，使它引用下一个元素，还需要编写一个方法测试它是否越过了集合的尾部。我们应该同时考虑这两个操作。在推进迭代器之前，我们需要知道当迭代器越过了列表的最后一个节点之后应该具有什么值。如果不想搞什么特殊，迭代器在这个时候很自然应该是 NULL 值，这也是最容易使用的值。注意，我们已经在默认构造函数中把迭代器初始化为 NULL，因此当我们用 NULL 提示越过集合尾部时，就会在这两种状态之间产生混淆。但是，对于当前的问题而言，这并不会造成什么麻烦。这个方法的代码如下：

```
❶ void scIterator::advance() {
   ❷if (current != NULL)
      ❸current = current->next;
  }
❹ bool scIterator::pastEnd() {
     return current == NULL;
  }
```

记住，我们只是用迭代器概念来解决原先的问题。我们并不需要复制 C++标准模板库的迭代器类的准确规范，因此无需使用相同的接口。在这个例子中，我们并非对++操作符进行重载，而是选择了一个称为 advance❶的方法，它判断当前的指针是否为 NULL❷，然后再把它推进到下一个节点❸。类似地，我发现创建一种特殊的“尾”迭代器并与之进行比较

是种很笨拙的做法，因此决定只选择一个称为 pastEnd❹的 bool 方法，用于确定是否已经遍历完了节点。

最后，我们需要一种方法获取当前所引用的 studentRecord 对象：

```
studentRecord scIterator::student() {
 ❶if (current == NULL) {
      studentRecord dummyRecord(-1, -1, "");
      return dummyRecord;
 ❷} else {
      return current->studentData;
   }
}
```

正如我们之前所做的那样，为了安全起见，如果指针的值为 NULL，我们就创建并返回一条哑记录❶。否则，我们就返回当前所引用的记录❷。这样我们就完成了 studentCollection 类的迭代器概念的实现。为了清晰起见，以下是 scIterator 类的完整声明：

```
class scIterator {
public:
   scIterator();
   scIterator(studentCollection::studentNode * initial);
   void advance();
   bool pastEnd();
   studentRecord student();
private:
   studentCollection::studentNode * current;
};
```

完成了所有的代码之后，我们可以用一个示例遍历对代码进行测试。下面我们实现平均成绩计算以进行比较。

```
scIterator iter;
int gradeTotal = 0;
int numRecords = 0;
❶ iter = sc.firstItemIterator();
❷ while (!iter.pastEnd()) {
     numRecords++;
   ❸gradeTotal += iter.student().grade();
   ❹iter.advance();
  }
  double average = (double) gradeTotal / numRecords;
```

这段代码使用了所有与迭代器相关的方法，因此可以对我们的代码进行很好的测试。我们调用 firstItemIterator 函数对 scIterator 对象进行初始化❶，调用 pastEnd

函数作为循环终止测试❷。我们还调用迭代器对象的 student 方法获取当前的 studentRecord 以便提取成绩❸。最后，为了把迭代器移动到下一条记录，我们调用了 advance 方法❹。当这段代码顺利运行时，我们可以合理地确信自己已经正确地实现了各个方法，而且对迭代器的概念有了坚实的理解。

高效遍历解决方案的分析

和以前一样，代码能够工作并不意味着这个事件的学习潜力就到此为止了。我们还应该仔细考虑完成了什么任务、它的正面效果和负面影响，并对我们所实现的基本思路的相应扩展进行思考。在这个例子中，我们可以认为迭代器的概念确实解决了客户代码对集合的低效遍历这个最初问题。一旦实现了迭代器之后，它的使用就变得非常优雅并容易理解。从负面的角度考虑，基于 recordAt 方法的低效方法显然要容易编写得多。在决定是否为一种特定的情况实现迭代器时，必须考虑遍历的发生频率、列表中一般会出现多少个元素等问题。如果很少进行对列表的遍历并且列表本身很短，那么低效问题很可能并不严重。但是，如果我们预期列表将会增长或者无法保证它不会增长，那么就应该使用迭代器方法。

当然，如果我们已经决定使用标准模板库的一个 list 类对象，就不需要再担心迭代器的实现难度这个问题，因为我们用不着自己实现它。下次再遇到类似的情况时，我们就可以使用 list 类，而不必感觉自己是在偷懒，也不必认为以后会在这方面遇到困难。因为我们已经对列表和迭代器进行了研究，理解了它们幕后的工作原理，即使自己从来没有研究过它们的实际源代码。

把话题再深入一步，我们可以考虑迭代器的更广泛应用以及它们可能存在的限制。例如，假设我们需要一个迭代器，不仅希望它能够高效地推进到 studentCollection 中的下一个元素，而且能够同样高效地退回到前一个元素。既然我们已经理解了迭代器的工作原理，就很容易明白在当前的 studentCollection 实现上是没有办法完成这个任务的。如果迭代器维护一个指向列表中某个特定节点的链（即 next 字段），把它推进到下一个节点只需要访问节点中的这个链。但是，撤回到前一个节点则要求反向遍历列表。我们可以采用双链表，每个节点维护两个分别指向前一个节点和下一个节点的指针。我们可以对这个思路进行归纳，开始考虑不同的数据结构以及它们可以向客户提供的高效的遍历类型或数据访问。例如，在前一章讨论递归时，我们简单地讨论了二叉树结构。是否存在标准的方法，允许客户代码以标准的形式对这种数据结构进行高效的遍历呢？如果不能，怎样对它进行修改以允许高效的遍历呢？对二叉树进行遍历的正确顺序是什么呢？考虑这样的类似问题可以帮助我们成为更优秀的程序员。我们不仅能够学习新的技巧，还能够了解不同

组件的优点和缺点。了解组件的优缺点可以帮助我们合理地使用组件。没有考虑到一种特定方法所存在的限制可能会导致悲惨的结果。对自己所使用的组件了解越多，发生这种事件的概率也就越低。

7.4 选择组件类型

正如我们在这些例子中所看到的那样，示例问题可以通过不同类型的组件来解决。一个模式可能表达了一种解决方案的思路，一种算法可能规划了思路的一种实现或者解决同一个问题的另一种思路，一种抽象数据类型可能封装了某个概念，类库中的一个类可能包含了一种抽象数据类型的完整的、经过测试的实现。如果它们都是对于解决我们的问题所需要的同一个概念的一种表达，那么我们怎样才能知道哪种组件类型放进我们的工具箱是最适合的呢？

一个主要的考虑是把组件集成到自己的解决方案需要多大的工作量。把一个类库链接到自己的代码常常是解决问题最迅速的方法，从一段伪码描述实现一种算法可能需要大量的时间。另一个重要的考虑是组件所提供的灵活性有多大。组件常常是以一种漂亮的、预包装的形式出现，但是当它集成到解决方案时，程序员发现虽然这个组件具有他所需要的大多数功能，但它并不能完成所有的任务。例如，也许一个方法的返回值格式不正确，需要额外的处理。如果坚持使用这个组件，在使用过程中可能会出现更多的问题，最后还是不得不放弃，只能从头寻找新的方案。如果程序员选择了一种位于更高概念层次的组件（如模式），最终的代码实现将会完美地适合需要解决的问题，因为它就是根据这个问题而创建的。

图 7.1 对这两个因素的相互影响进行了总结。一般而言，来自类库的代码马上就能被使用，但它无法被直接修改。它只能通过间接的方式修改，或者使用 C++模板，或者让解决问题的代码实现本章前面所提到的策略模式之类的东西。另一方面，模式所表示的东西可能仅仅是个思路（如“这个类只能具有 1 个实例”），它提供了最大的实现灵活性，但是对于程序员而言则需要大量的工作。

当然，这并不是一个基本的指导方针，每个人所面临的情况可能各不相同。也许我们从类库中所使用的类在自己的程序中位于相当低的层次，它的灵活性并不重要。例如，我们可能想自己设计一个集合类，包装了类似 list 这样的基本容器类。由于 list 类所提供的功能相当广泛，因此即使我们必须对这个集合类的功能进行扩展，预计作为底层容器的 list 类也完全能够胜任。在使用模式之前，也许过去已经实现了一个特定的模式，我们可以对以前所编写的代码进行适配，这样就不需要创建太多的新代码。

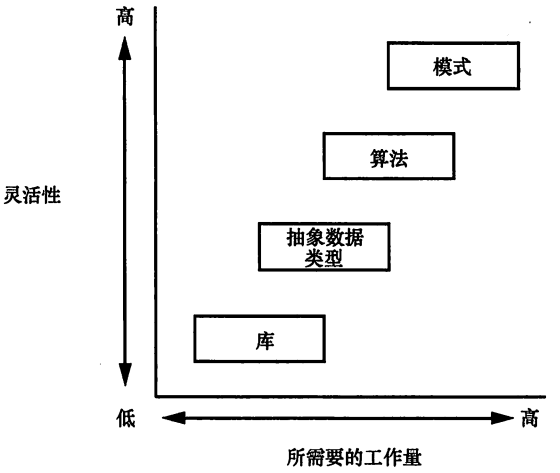

图 7.1 组件类型的灵活性与所需要的工作量

在使用组件方面的经验越丰富，对于选择正确的组件就会有更大的自信。在积累足够的经验之前，可以把灵活性和所需工作量之间的权衡作为粗略的指导方针。对于每种特定的情况，可以提出下面这几个问题：

- 能不能直接使用这个组件？还是需要额外的代码才能让它应用于自己的项目？
- 我是否确信已经从各个方面理解了问题，或者理解了与这个组件相关联的问题，并且确定它在未来也不会发生变化？
- 通过选择这个组件，是不是能够扩展我的编程知识？

这些问题的答案可以帮助我们评估选择某个组件所需要的工作量以及自己能够从每种可能的方法中获得多大的益处。

组件选择的实例

现在我们已经理解了基本的思路，下面可以通过一个简单的例子来说明具体的细节了。

对某些数据进行排序，但其他数据保持不变

一个项目要求我们对一个 studentRecord 对象数组按成绩进行排序，但是存在一个难点：这个程序的另一部分使用-1 这个特殊的成绩值表示那些无法移动记录的学生。因此，尽管所有其他记录必须移动，但那些成绩值为-1 的记录必须保留在原先的位置。最终所产生的结果是一个排好序的数组，但其间散布着一些成绩值为-1 的记录。

这是一个需要技巧的问题，我们可以尝试用多种方法来解决这个问题。为了简单起见，我们把选项减为2个：其一是选择一种算法，例如像插入排序这样的算法并对它进行修改，忽略那些成绩值为-1的studentRecord对象。其二是想出一种方法，用qsort库函数来解决这个问题。这两个选择都是可行的。由于我们已经熟悉了插入排序的代码，在它的里面插入几条if语句，显式地检查并跳过那些成绩值为-1的记录应该不会太困难。让qsort为我们完成工作就需要一些变通。我们可以把具有真正成绩值的学生记录复制到一个单独的数组中，用qsort对它们进行排序，然后再复制回原先的数组，并保证在复制时不会覆盖原先成绩值为-1的记录。

让我们对这两个选项进行分析，观察组件类型的选择是怎样影响最终代码的。我们首先从算法组件开始，编写经过修改的插入排序算法来解决这个问题。和往常一样，我们将分几个阶段来解决这个问题。首先，我们通过去掉-1成绩这个阶段性问题来削减这个问题，对studentRecord对象数组进行排序时不考虑任何特殊规则。如果sra是包含了arraysize个studentRecord类型的对象数组，它的代码应该如下所示：

```
int start = 0;
int end = arraySize - 1;
for (int i = start + 1; i <= end; i++) {
   for (int j = i; j > start && ❶sra[j-1].grade() > sra[j].grade(); j--) {
    ❷studentRecord temp = sra[j-1];
      sra[j-1] = sra[j];
      sra[j] = temp;
   }
}
```

这段代码与整数的插入排序非常相似。唯一的区别是它在执行比较时调用了grade方法❶，另外还更改了用于交换空间的临时对象的类型❷。这段代码能够顺利完成任务，但是对它以及本节后面的代码段进行测试的时候，有一个需要警惕的地方：正如以前所编写的那样，studentRecord类会对数据执行验证，它不会接受-1作为成绩值，因此需要进行必要的修改。现在，我们就可以完成这个版本的解决方案了。我们需要让插入排序忽略成绩值为-1的记录。这个任务并不像听上去那么简单。在基本的插入排序算法中，我们总是在数组中交换相邻的位置，如上面代码中的j和j - 1。但是，如果我们让有些记录的成绩值保留为-1，那么需要与当前记录进行交换的下一条记录的位置可能相隔甚远。

图7.2用一个例子描述了这个问题。它显示了最初配置下的数组，并用箭头提示第1条记录需要被交换到的位置，它们并不是相邻的。而且，最后一条记录（表示Art）最终将从位置[5]交换到位置[3]，然后再从[3]交换到[0]，因此对这个数组排序所进行的所有交换都涉

及到非相邻的记录（至少对于我们所排序的那些记录是这样的）。

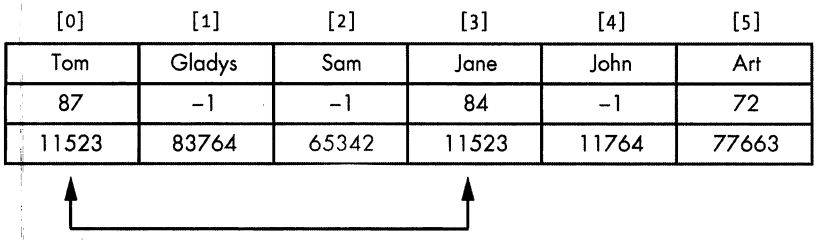

图 7.2 修改后的排序算法中需要被交换的记录之间的任意距离

在考虑怎样解决这个问题时，我设法寻找一个类比。我在处理链表的问题的选择中找到了一个类比。在许多链表算法中，我们在链表遍历时不仅需要维护一个指向当前节点的指针，还需要维护一个指向前一个节点的指针。因此在循环体结束的时候，我们经常把当前节点指针赋值给前一节点指针，然后再把当前节点指针指向下一个节点。这个例子也需要类似的做法。当我们按照线性顺序遍历这个数组寻找下一条“真正的”学生记录时，还需要追踪前一条“真正的”的学生记录。把这个思路投放到实践中就产生了如下的代码：

```
for (int i = start + 1; i <= end; i++) {
 ❶if (sra[i].grade() != -1) {
    ❷int rightswap = i;
      for (int leftswap = i - 1;
         leftswap >= start
         && (sra[leftswap].grade() > sra[rightswap].grade()
          ❸|| sra[leftswap].grade() == -1);
         leftswap--)
      {
       ❹if(sra[leftswap].grade() != -1) {
            studentRecord temp = sra[leftswap];
            sra[leftswap] = sra[rightswap];
            sra[rightswap] = temp;
          ❺rightswap = leftswap;
         }
      }
   }
}
```

在基本的插入排序算法中，我们反复地把未排序的元素插入到数组中一块不断增长的已排序区域中。外层的循环选择下一条需要被放到排序区的未排序元素。在这个版本的代码中，我们首先在外层循环体中判断位置 i 的成绩值是不是-1❶。如果是，我们就简单地跳

到下一条记录，保留这个位置不变。当我们确定位置 i 的学生记录可以被移动时，就把 rightswap 初始化为这个位置❷。然后我们就进入内层循环。在基本的插入排序算法中，内层循环的每次迭代都把一个元素与它相邻的元素进行交换。但是，在这个版本的插入排序中，由于我们让成绩值为-1 的学生记录保持不动，所以只有当位置 j 的学生记录的成绩值不是-1 时才执行交换❹。然后，我们在 leftswap 和 rightswap 这两个位置之间执行交换并把 leftswap 赋值给 rightswap❺，如果还有要执行的交换就设置下一次交换。最后，我们必须修改内层循环的终止条件。正常情况下，插入排序的内层循环是在到达了数组的前端或者找到了小于需要被插入值的元素的时候终止。在这个例子中，我们必须用逻辑或操作符创建一个复合条件，使循环能够跳过成绩值为-1 的记录❸。（由于-1 小于所有合法的成绩值，因此会永久地停止循环。）

这段代码解决了我们的问题，但它很可能会散发出某种“不良的气味”。标准的插入排序算法很容易理解，尤其是当我们理解了它的主旨时。但是，这个经过修改的版本就很难读懂，如果我们想在以后还能看懂这段代码，很可能需要添加几条注释。也许可以对它进行重构，但我们先试试用其他方法来解决这个问题并观察其结果。

我们所需要的第一样东西就是一个在 qsort 中使用的比较函数。在这个例子中，我们将比较两个 studentRecord 对象，并且这个函数将把一个成绩值减去另一个成绩值：

```
int compareStudentRecord(const void * voidA, const void * voidB) {
   studentRecord * recordA = (studentRecord *) voidA;
   studentRecord * recordB = (studentRecord *) voidB;
   return recordA->grade() - recordB->grade();
}
```

现在，我们就可以对记录进行排序了。我们将分为 3 个阶段完成这项任务。首先，我们把所有成绩值不是-1 的记录复制到第 2 个数组，元素之间没有空隙。接着，我们调用 qsort 对第 2 个数组进行排序。最后，我们把第 2 个数组的记录复制回原来的数组，跳过那些成绩值为-1 的记录。最终的代码如下所示：

```
❶ studentRecord sortArray[arraySize];
❷ int sortArrayCount = 0;
  for (int i = 0; i < arraySize; i++) {
   ❸if (sra[i].grade() != -1) {
        sortArray[sortArrayCount] = sra[i];
        sortArrayCount++;
    }
  }
❹ qsort(sortArray, sortArrayCount, sizeof(studentRecord), compareStudentRecord);
```

```
❺ sortArrayCount = 0;
❻ for (int i = 0; i < arraySize; i++) {
    ❼if (sra[i].grade() != -1) {
        sra[i] = sortArray[sortArrayCount];
        sortArrayCount++;
    }
}
```

尽管这段代码的长度和前面那个解决方案差不多，但它更加简明易懂。我们首先声明第 2 个数组 sortArray❶，它的长度与原先的数组相同。sortArrayCount 变量被初始化为 0❷。在第 1 个循环中，我们将用这个变量追踪检查有多少条记录已经被复制到第 2 个数组中。在这个循环的内部，每次遇到一条成绩值不是-1 的记录时❸，我们就把它赋值给 sortArray 中的下一个空位置并将 sortArrayCount 的值增加 1。当这个循环结束时，我们就对第 2 个数组进行排序❹。sortArrayCount 变量被重置为 0❺。我们将在第 2 个循环中用它追踪检查有多少条记录已经从第 2 个数组复制回原先的数组。注意，第 2 个循环对原先的数组进行遍历❻，寻找需要被填充的位置❼。如果我们用其他方法来完成这个任务，可以尝试对第 2 个数组进行遍历，并把记录推回到原来的数组。那样，我们将需要一个双重循环，其中内层循环搜索原先的数组中下一个具有真正成绩值的位置。这是问题的难易程度取决于它的概念化层次的又一个例子。

比较结果

这两个解决方案都可以完成任务并且都采用了合理的方法。对于大多数程序员而言，对插入排序进行修改并在排序时使部分记录保持不动的第 1 个解决方案很难编写和读懂。但是，第 2 个解决方案似乎有些低效，因为它需要把数据复制到第 2 个数组并复制回来。下面对这两种算法进行简单的分析。假设我们对 10000 条记录进行排序，如果需要排序的次数极少，那就无需太关注效率问题。我们无法确切地知道 qsort 所采用的底层算法是什么，但是对于通用目的的排序，最坏的情况是要执行 1 亿次的记录交换，最佳的情况只需要执行 13 万次。不管实际的交换次数是这个范围内的哪个数字，来回复制 10000 条记录相对于排序而言并不会对性能产生非常大的影响。另外，还必须考虑 qsort 所采用的排序算法可能比我们所采用的简单排序更为高效，这样使第 1 个解决方案不需要把数据来回复制到第 2 个数组的优点也化为乌有。

因此在这个场景中，使用 qsort 的第 2 种方法要更好一点。它更容易实现、更容易读懂，因此也更容易维护。并且，我们可以预期它的执行效率不逊于第 1 个解决方案，甚至更胜一筹。第 1 个解决方案的最大优点是我们可以把学会的技巧应用于其他问题，而第 2 个解决方案由于过于简单而缺乏这一点。作为基本规则，当我们处在需要最大限度地提高

自己的编程能力的阶段时，应该优先选择高层次的组件，例如算法或模式。当我们处在需要最大限度地提高编程效率（或者时间期限非常紧张）的阶段时，应该优先考虑低层次的组件，尽可能选择预创建的代码。当然，如果时间允许，可以尝试不同的方法，就像我们刚刚所完成的那样，这样可以得到最大的收获。

7.5 习题

尽可能多地对组件进行试验。一旦掌握了怎样学习新组件，将会极大地提高自己的编程能力。

7.1 对策略模式的一个反对意见是它需要暴露类的一些内部实现，例如类型。修改本章前半部分的“班长”程序，使策略函数都存储在这个类中，并通过传递一个代码值（例如，一种新的枚举类型）来进行选择，而不是传递策略函数本身。

7.2 重新编写第4章的 `studentCollection` 函数(`addRecord` 和 `averageRecord`)，不需要直接实现一个链表，而是使用一个来自 C++库的类。

7.3 考虑一个 `studentRecord` 对象的集合。我们想要根据学生编号寻找一条特定的记录。把学生记录存储在一个数组中，根据学生编号对数组进行排序，并研究和实现插值搜索算法。

7.4 对于习题 7.3 的问题，通过一个抽象数据类型来实现一个解决方案。这个抽象数据类型允许存储任意数量的数据项，并可以根据键值提取单独的记录。对于能够根据一个键值高效地存储和提取数据项的结构，它用基本术语表示就是符号表，符号表思路的常见实现是散列表和二叉搜索树。

7.5 对于习题 7.3 的问题，实现一个使用 C++库类的解决方案。

7.6 假设我们正在完成一个项目。在这个项目中，一个特定的 `studentRecord` 可能需要添加下面这些数据之一：学期论文标题、注册年份或表示该学生是否对班级进行审计的 `bool` 值。我们并不想在基本的 `studentRecord` 类中添加所有这些字段，因为它们在大多数情况下并不会被使用。我们的第一个想法是创建 3 个子类，每个分别包含了上面这 3 个数据字段之一。这 3 个子类的名称应该诸如 `studentRecordTitle`、`studentRecordYear` 和 `studentRecordAudit`。然后，我们得知有些学生记录需要包含上述 3 个数据字段的 2 个甚至全部 3 个。

为每种可能出现的情况分别创建一个子类显然是不切实际的。寻找一种能够处理这种情况的设计模式，并实现一个解决方案。

7.7 为习题 7.6 所描述的问题开发一个解决方案，不要使用自己所发现的模式，而是用 C++库类来解决这个问题。不要把注意力集中在前一个问题所描述的 3 个特定数据字段，而是创建一个通用的解决方案：这个版本的 `studentRecord` 类允许向特定对象添加任意数量的额外字段。例如，`sr1` 是个 `studentRecord` 对象，我们允许客户代码调用 `sr1.addExtraField("Title", "Problems of Unconditional Branching")`，以后再调用 `sr1.retrieveField("Title")` 将返回“`Problems of Unconditional Branching`”。

7.8 自行设计：针对一个自己已经解决的问题，使用不同的组件再次解决它。对结果进行分析，并将其与原来的解决方案进行比较。

第 8 章

培养程序员的思维

现在是时候把我们在前面各章所学习的知识揉和到一起，完成从雏鸟初飞的新手到能够熟练解决问题的程序员的蜕变。

在前面各章中，我们解决了各个不同领域的问题。我相信对这些领域的问题的研究最有益于程序员的成长。当然，我们需要学习的东西还有很多，许多问题需要本书未能覆盖的技巧才能解决。在本章中，我们将尽可能完整地讨论通用的问题解决方案，利用我们在前面各章的探索旅程中所获得的知识，开发出一个总体计划来解决任何编程问题。尽管我们把它称为总体计划，但实际上它只是一个非常特定的规划：它将成为我们自己的规划，而不是其他人的。我们还将讨论程序员用于增长知识和提升技能的许多方式。

8.1 创建自己的总体计划

在第 1 章中，我们学习了解决问题的第一个原则是“总是要有计划”。更准确的说法是

总是遵循自己的计划。我们应该创建一个总体计划，最大限度地发扬长避短，然后把这个总体计划应用于自己必须解决的每个问题中。

在多年的教学生涯中，我看到过很多能力不同的学生。我不能简单地说有些程序员比其他程序员更有能力，虽然事实可能确实如此。即使是在相同能力水平的程序员之间，也存在很大的区别。我经常不可思议地看到以前学习得很挣扎的学生很快精通了某种特定的技巧，或者在其他领域天赋卓然的学生在一个新领域却暴露出明显的弱点。就像不存在两个完全相同的指纹一样，没有两个大脑是完全相同的，对于一个人来说非常容易的一堂课对于另一个人来说可能非常困难。

假设读者是一位美式足球教练，正在制订下一场比赛的进攻计划。由于伤病的原因，无法确定两名四分卫谁能够首发登场。这两名四分卫都具有高度的职业素养，但是和所有人一样，他们也有各自的优点和缺点。为一位四分卫所制订的完美比赛计划套用于另一位四分卫身上却可能带来糟糕的结果。

在创建总体计划时，教练需要根据队中的四分卫进行排兵布阵。为了实现最大的获胜机率，需要制订一个计划，既要认识到自己的优势，也要明白自己的弱点。

8.1.1 扬长避短

在制订自己的总体计划时，关键的步骤是认识到自己的优势和弱点。这并不困难，但它需要花费精力并且需要一个公平的自我评估。为了从错误中获益，不仅需要在程序中所出现的地方修正它们，还必须对它们进行关注，至少是在大脑里，最好是记录在文档中。通过这种方式，可以发现在其他情况下可能错失的行为模式。

下面将描述两种不同类型的弱点：编码弱点和设计弱点。编码弱点是指在实际编写代码时可能反复犯错的领域。例如，许多程序员在编写循环的时候，经常会出现迭代次数多 1 次或少 1 次的情况。这个错误称为栅栏柱错误，它取材于一个古老的难题，就是建造一条总长 50m 的栅栏并且每根栅柱之间相隔 10 英尺，一共需要几根柱子？大多数人的第一反应是 5，但是如果仔细考虑，答案应该是 6，如图 8.1 所示。

大多数编码弱点出现在由于程序员编写代码过于迅速或者缺乏充分准备而导致语义错误的情况下。反之，设计弱点在问题的解决或设计阶段经常出现。例如，我们可能不知道该怎么入手或者不知道怎么把以前所编写的子程序集成到一个完整的解决方案中。

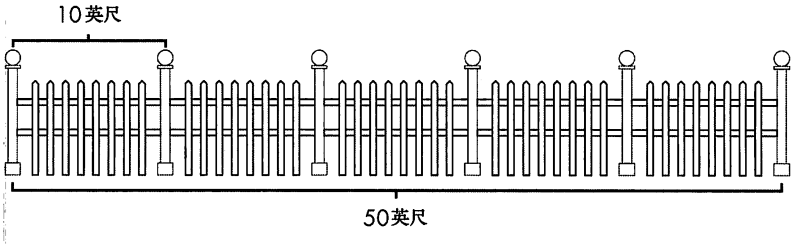

图 8.1 栅栏难题

尽管这两种类型的弱点存在一些重叠，但它们会导致不同类型的问题，因此必须按照不同的方式予以解决。

针对编码弱点的计划

在编写程序的时候，最令人气恼的事情莫过于花了几个小时的时间追踪一处语义错误，结果却发现只是一个非常简单而且很容易修正的错误。没有任何东西是完美的，因此没有办法完全消除这样的情况，但是优秀的程序员将会尽他所能避免相同的错误再次发生。

有一位程序员已经厌倦了 C++编程中可能最为常见的语义错误：误用赋值操作符（=）代替了相等操作符（==）。由于 C++的条件表达式的结果是整数，而不是严格的布尔值，因此下面这样的语句在语法上是合法的：

```
if (number = 1) flag = true;
```

在这种情况下，整数值 1 被赋值给 number，然后 1 这个值成为了条件语句的结果，C++把它当作 true 处理。显然，程序员的意图其实是：

```
if (number == 1) flag = true;
```

被这类错误的屡次发生搞得心烦气躁之后，这位程序员告诫自己用另一种方式来编写相等性测试，让数字值出现在左边。例如：

```
if (1 == number) flag = true;
```

通过这种做法，如果他不小心再次犯了上面这个错误，1 = number 这个表达式将不再是合法的 C++表达式，因此会产生语法错误，会在编译时被捕捉。原来的错误在语法上是合法的，因此它只是个语义错误，在编译时可能会被捕捉，但也可能根本不会被捕捉。由于我自己也曾经多次犯过这个错误（有时候在查找这个错误时会急得发疯），因此也采用了这种方法，把数字值放在相等操作符的左边。采用这种做法之后，我发现了一些奇怪的事情。由于它与我平时所使用的风格相悖，因此在编写条件语句时把数字值放在左边会让我的思维暂时停顿。我会这么思考：“我需要记住把数字值放在左边，这样在误用了

赋值符时就能发现这种情况”。正如读者所料，把这种想法在脑子里过一遍之后，绝不会再错误地使用赋值操作符，而是能够正确地使用相等操作符。现在，我不再把数字值放在相等操作符的左边，但在编写条件表达式时仍然会习惯性地停顿一下，使上面这个想法再过过脑子，这样就不会再犯这种错误了。

通过这个事情，我所得到的一个经验就是：首先要意识到自己在编码层次上的弱点，然后才能有效地避免它们。这是好消息，坏消息是我们必须通过实践才能认识到自己的编码弱点。关键的技巧是让自己知道为什么会犯某个特定的错误，而不仅仅是修正这个错误并继续下一步工作。这可以帮助我们确认自己有没有遵循的某个基本原则。例如，我们编写了下面这个函数，计算一个整数数组中所有正数的平均值：

```
double averagePositive(int array[ARRAYSIZE]) {
   int total = 0;
   int positiveCount = 0;
   for (int i = 0; i < ARRAYSIZE; i++) {
      if (array[i] > 0) {
         total += array[i];
         positiveCount++;
      }
   }
 ❶return total / (double) positiveCount;
}
```

初看上去，这个函数并没有什么问题。但是经过仔细观察，还是可以看到它存在一个问题。如果数组中没有任何正数，当循环结束时 positiveCount 的值将是 0，这将导致在函数结束时执行除零运算❶。由于这是浮点除法，因此程序可能不会实际崩溃，而是产生某种奇怪的行为，这具体取决于这个函数的返回值在整体程序中是怎样被使用的。

如果我们很快就设法运行了这段代码，并且发现了这个问题，可能会添加一些代码，处理 positiveCount 为零的情况，并继续下一步工作。但是，如果想完善自己的编程能力，就应该问问自己犯了什么错误。当然，这个特定的问题是没有考虑到除零的可能性。但是，如果分析只是到此为止，并不会对未来提供多大的帮助。显然，此时应该考虑分母可能为零的其他情况，但这也不算一种非常常见的情况。反之，我们应该问问自己是否违背了什么基本原则。答案是：我们要坚持寻找那些可能导致代码失败的特殊情况。

考虑到这个基本原则之后，就很容易看到我们所犯错误的模式，因此在将来很容易捕捉到这类错误。问自己“这里是不是存在除零错误的可能性”远不如问自己“这个数据存在什么特殊情况”更为有用。提出作用面更宽的问题，除了能够想到不要出现除零运算之外，还会迫使自己考虑空数据集、超出预期范围的数据等问题。

针对设计弱点的计划

设计弱点需要一种不同的方法才能克服。但是，第一个步骤仍然是一样的，就是要认识到自己的弱点所在。很多人在这个步骤上存在问题，因为他们并不愿意对自己采取批评态度。人们总会想方设法隐瞒自己的失败。就像接受工作面试时，当一位面试官问你最大的弱点是什么时，你很可能会回答一些不会对面试结果产生影响的弱点，而不是坦率地承认自己的真正缺点。但是，就像超人也受制于氪星石一样，就算是最优秀的程序员也存在真正的弱点。

下面是程序员弱点的一个示例列表(当然并不完整)，读者可以看看自己是否符合其中的几条。

过于复杂的设计

存在这个弱点的程序员所创建的程序常常具有过多的组成部分，或者具有过多的步骤。虽然程序能够完成任务，但它们无法让自己充满信心，就像穿上去的衣服一扯线头就会全部散架一样。很显然，它们是非常低效的。

不知如何着手

这种类型的程序员具有高度的惰性。也许是由于在解决问题上缺乏信心，也可能生来就是慢性子，这类程序员花费了太多时间考虑怎么开始解决问题。

疏于测试

这类程序员不喜欢对自己的代码进行正式的测试。这样的代码在一般情况常常能够很好地完成任务，但是面对特殊情况时常常会导致失败。还有一些情况下，这样的代码能够顺利地完成任务，但是对于程序员没有进行测试的大型问题，它就难以表现出应有的适应能力。

过分自信

自信是件好事，本书的目标之一就是培养读者的自信心。但是，过分自信和不够自信一样并非好事。过分自信会通过各种方式表现出来。过分自信的程序员可能会尝试一种超出需要的更复杂解决方案，或者在非常短的时间内就匆匆完成一个项目，导致粗率、缺陷丛生的程序产生。

脆弱领域

这种类型的弱点可谓五花八门。有些程序员一直在顺利地工作，但在遇到了某些概念后突然变得不知所措。以本书前面各章所讨论的话题为例，大多数程序员在面对某个领域时，就算完成了所有的习题，他们在这个领域的信心也要比在其他领域弱得多。例如，有些程序员会迷失于指针程序中；或者递归的概念会把有些程序员的脑子搞混。有些程序员在设计详尽的类时会遇到困难。这并不是说这些程序员就没有办法应付这些问题，但对他们而言这些是非常艰巨的任务，就像在泥地里开车一样。

我们可以通过不同的方法暴露自己的主要弱点。一旦认识到自己的弱点之后，就很容易针对它们制订计划。例如，对于经常忽略测试的程序员，在制订每个模块的编写计划时，可以明确地把测试作为必须完成的部分，在完成测试之前不能开始下一个模块的设计。或者，也可以考虑一种称为“测试驱动的开发”的设计用法。在这种惯用法中，首先编写测试代码，再编写填充这些测试的其他代码。对那些迟迟不能入手的程序员，可以采用问题的分治或削减原则，一旦他觉得可以就开始编写代码，当然还要知道将来可能需要对这些代码进行重写。对于那些常常设计得过于复杂的程序员，可以在总体计划中增加一个明确的重构步骤。关键在于，不管程序员的主要弱点是什么，它们只不过是项目成功完成的道路上的绊脚石而已。

根据自己的优点制订计划

根据弱点制订计划在很大程度上是为了避免错误。但是，良好的计划并不仅仅是为了避免错误。它还涉及到根据自己的当前能力以及可能受到的约束，尽可能实现最佳结果。这意味着我们还必须根据自己的优点制订总体计划。

读者可能觉得本节的内容不适合自己，至少目前为止还不适合。不管怎样，如果读者已经开始阅读本书，就有可能成为一名程序员。读者可能觉得自己在当前阶段还谈不上有任何优点，但事实上还是有的，即使自己并没有意识到它们。下面是一些常见的程序员优点的列表，当然并不完整。我对每个优点提供了描述和提示，以帮助读者判断自己是否具有这些优点：

细心

这种类型的程序员能够预料到特殊情况，在潜在的性能问题出现之前就预感到它们，而且绝不会让整体情况掩盖那些必须精心处理的细节，而这些细节又往往是实现完整和准确的解决方案所必需的。具有这个优点的程序员倾向于在编写代码之前先在纸上测试他们的计划，他们会小心细致地编写代码，并且经常进行测试。

快速学习能力

具有快速学习能力的程序员能够很快学会新的技巧，无论是一种已经熟悉的语言中的一项新技巧还是学习一个新的应用程序框架。这种类型的程序员享受学习新事物的挑战，可能会根据这个喜好来选择项目。

快速编码能力

具有快速编码能力的程序员无需很长时间就可以根据一本参考书捣鼓出一个函数。到了开始打字的时间，不需要特别地费劲，代码就会从指尖迅速涌出，并且其中很少出现语法错误。

永不放弃

对于有些程序员而言，讨厌的程序缺陷就像无法回避的个人遭遇一样。如同程序戴着皮革手套扇了程序员一个巴掌，然后轮到程序员对此做出回应。这类程序员始终头脑冷静、意志坚定，不会被挫折所击倒。他们坚信只要付出足够的努力，必将取得最后的胜利。

超级问题解决专家

假如读者在阅读本书时还不是一位超级问题解决专家，但是在了解了一些指导方针之后，觉得做所有的事情时都变得得心应手。那么，具有这种能力的程序员在刚刚接触一个问题的时候就会开始思考潜在的解决方案。

完美主义者

对于这类程序员而言，一个工作程序就像一件精妙的玩具。完美主义者绝不会丧失让计算机按照他的命令行事的激情，并且喜欢想方设法找点事情让计算机去做。在某种意义上，完美主义意味着不断向工作程序添加更多的功能，这种症状称为爬行功能主义。在他们眼里，也许可以对程序进行重构以提高性能，也许可以让程序在程序员或用户面前显得更精巧。

很少有程序员能够同时拥有上面所说的多个优点。事实上，有些优点会相互抵销。但是，每个程序员都有自己的优点。如果读者觉得自己不符合上面所说的任何一条，也只是意味着对自己还不够了解，或者其优点并不属于上面所提到的这几种类型。

一旦确认了自己的优点，就需要在总体计划中利用它们。假设读者具有快速编码能力，很显然它可以使任何项目更快地到达终点。但是，怎样才能以系统的方式利用这个优点呢？在正式的软件工程中，有一种方法称为快速原型法，就是在一开始编写一个程序的时候并没有深入的计划，需要通过连续的迭代予以完善，直到最终的结果能够满足问题需求。作为快速编码者，可以尝试采用这种方法：有了一个基本的思路之后就可以开始编写代码，用粗略的原型来指导最终程序代码的设计和开发。

如果读者具有快速学习能力，在每个项目开始的时候应该寻访新的资源或技巧来解决当前问题。如果既不具备快速学习能力，但也不会轻易被挫折所击垮，那么在项目开始的时候可以从最困难的部分入手，给自己最多的时间来处理它们。

因此，不管自己拥有何种优点，要保证在编程时利用它们。设计自己的总体计划，尽可能地把时间留给自己最擅长的事情。通过这种方式，读者在编程时不仅能够产生最好的结果，还将体会到最多的乐趣。

8.1.2 制订总体计划

让我们观察创建总体计划的一个实例。这个计划的组成部分包括自己已经掌握的所有问题解决技巧，再加上对自身的优点和弱点的分析。我将使用自己的优点和弱点作为例子。

在问题解决技巧方面，我用了本书所讨论的所有技巧，但尤其钟爱“削减问题”技巧，因为这种技巧能够让我感觉到自己向最终的目标不断迈出坚实的步伐。如果目前还无法编写满足完整规范的代码，可以先忽略部分规范，直到有信心完成剩余的内容为止。

我最大的编码弱点是过于认真。我喜欢编程，因为喜欢看到计算机按照自己的命令行事。有时候，应该分析自己所编写的东西的正确性时，我会考虑：“直接让它运行吧，看看会发生什么。”这种做法的危险在于程序可能会失败。虽然程序看上去似乎很成功，但是它并没有覆盖所有的特殊情况。或者它虽然成功，但并不是我应该编写的最佳解决方案。

我喜欢优雅的程序设计，希望程序能够很方便地被扩展和复用。当我编写大型项目时，常常花费大量的时间开发其他途径的设计方案。总体而言，它是一个良好的能力，但有时这会导致我把过多的时间花在设计阶段，没有留下太多的时间实现最终所选择的设计。另外，这也会导致解决方案的过度设计。也就是说，有时候设计的解决方案会比实际所需要的解决方案更优雅、更容易扩展并且更健壮。由于每个项目的时间和金钱都是有限的，因此最佳的解决方案必须同时兼顾程序的质量和资源的节约。

我觉得自己最大的编程优点就是能够很快领会新概念并且热爱学习。虽然有些程序员喜欢一直使用相同的技巧，但我喜欢在项目完成时能够学到新东西，并且总是很乐于接受这类挑战。

有了这些思路之后，下面是我对一个新项目的总体计划。

为了弥补我的主要弱点，我严格地控制自己在设计阶段所花费的时间，或者说控制了在设计阶段所考虑的不同设计方案的数量。对于某些读者而言，这好像是种危险的想法。在进入编码阶段之前，难道不应该尽可能地多花点时间在设计阶段吗？大多数项目失败的原因难道不是因为在前期所花的时间太少从而导致后期的一连串妥协吗？这些顾虑当然是对的，但是我现在并不是为软件开发创建一条通用的指导方针，而是为自己制订处理编程问题的总体计划。我的弱点是过度设计而不是设计不足，因此控制设计时间这个规则对我而言是合理的。对于其他程序员而言，这样的规则很可能是灾难。有些程序员可能需要一个规则迫使他们把更多的时间花在设计上。

完成最初的分析之后，我就开始考虑这个项目是否有机会让自己学习新的技巧、库等知识。如果答案是肯定的，我就打算编写一个小型的测验台程序，对这些新技巧进行试验，然后再把它们吸收到自己所开发的解决方案中。为了克服过于认真这个弱点，在完成每个

模式的编码时，可以添加一个简单的代码回顾步骤。但是，这并不是我的意愿所在。当我完成每个模块时，希望继续前进并让它实际运行。单纯地希望我在这个时候能够停下来就像在一个肚子饿得咕咕叫的人身边放上一袋打开的薯片，然后惊奇地发现这袋薯片被吃光了。在制订克服程序员弱点的计划时，不要让程序员跟自己的直觉做斗争。如果我创建了两个版本的项目：一个是原始的任由我处理的版本，另一个是经过优化的准备发行的版本。如果允许我对第一个版本按照自己的意愿行事，但是在经过完全的验证之前，不要把它的代码吸收到另一个优化的版本中，这样无疑更容易克服自己的弱点。

8.2 处理任何问题

制订了总体计划之后，我们就可以开始动手了。这正是本书所讨论的内容：从一个问题（任何问题）开始，寻找一种解决方案的方法。在前面各章中，问题描述使我们可以一开始把目光瞄准某个方向。但是在现实世界中，大多数程序并没有阐述需要使用数组或递归，或者要求把程序的一部分功能封装到一个类中。反之，程序员必须把这些决策作为问题解决过程的组成部分。

一开始，很少有需求能够让问题看上去变得更简单。不管怎样，设计需求是一种约束条件，难道约束条件不会使问题变得更困难吗？虽然这是对的，但不可否认的是所有的问题都有约束条件，只不过有些约束条件能够很明确地看到，有些则不是非常明显。例如，一个特定的问题没有明确表示是否需要动态分配的内存，并不意味着在做出这个决定时不会受到影响。对于一些更宽泛的约束条件，无论是关于性能、可修改性、开发速度还是其他方面的约束条件，一旦做出了错误的设计选择，就会导致难以满足甚至无法满足这些约束条件。

想象有一帮朋友要你选择一部让所有人一起观看的电影。如果一位朋友明确表示想看喜剧片，另一位朋友不喜欢老式电影，还有一位朋友列出了 5 部她刚刚看过的电影并表示不想重复观看。这些约束条件会导致你的选择变得困难。但是，如果每个人所提的建议是“只要挑部好看的就行”，你所面临的选择就会更加困难，很可能会挑选出一部至少有一位朋友根本不喜欢看的电影。

因此，按照更大、更广泛的定义，约束条件弱的问题常常是最难解决的。但是，它们可以通过本书所描述的问题处理技巧来解决，只是需要花费更多的时间。掌握了这些技巧并且制订了总体计划之后，就有能力解决任何问题。

为了证明刚才所讨论的观点，我打算带领读者完成绞刑者程序的第一个步骤。它是一种经典的儿童游戏，不过在此基础上我增加了一个变化。

在阅读问题描述之前，我们首先回顾一下这个游戏的基本规则。第 1 位玩家选择一个

单词并告诉第2位玩家这个单词包含了几个字母。接着，第2位玩家猜一个字母。如果该字母确实是在这个单词之中，第1位玩家就说出这个字母出现在单词中的位置。如果这个字母在单词中的出现次数不止1次，那么必须说明它所有的出现位置。如果这个字母并没有出现在这个单词之中，第1位玩家就在他所绘制的一幅绞刑者图画中添上一笔。如果第2位玩家猜中了这个单词的所有字母，那他就获得了胜利。但是，如果第1位玩家画完了绞刑者，他就获得了胜利。关于画完这个绞刑者一共需要多少笔，存在不同的规则。因此，换用更通俗的说法，两位玩家在游戏之前先约定第2位玩家猜错多少次就算第1位玩家获胜。

既然我们已经了解了基本规则，现在可以观察特定的问题，包括我所增加的那个具有挑战性的变化。

8.2.1 问题：绞型者作弊程序

编写一个程序，它基于文本的猜字游戏（也就是说，不需要实际绘制绞刑者，只需要记录不正确的猜测数量）的玩家1。玩家2通过指定待猜测的单词的长度以及输掉游戏时的猜错次数，设置游戏的难度。

我所增加的那个变化是程序可以进行作弊。它并不是在游戏开始时就实际挑选一个单词。程序可以避免挑选单词，只要当玩家2失败时，程序就能够显示一个与玩家2所提供的所有信息匹配的单词。所有猜中的字母必须出现在正确的位置上，猜错的字母不能出现在最终的单词中。当游戏结束时，玩家1（程序）会告诉玩家2这个单词已经被选好。因此，玩家2永远无法证明程序是在作弊，只不过发现自己的获胜机会很渺茫而已。

按照现实世界的标准，这并不是一个非常复杂的问题，但它足以说明当我们处理一个指定了结果但没有指定标准解决方法的编程问题时所面临的各种情况。根据问题描述，我们可以打开自己的开发环境，从十几个不同的情况挑选一个开始编写代码。当然，这是错误的方法，因为我们在开发程序之前总要制订一个计划。因此，我需要把自己的总体计划应用于这个特定的例子中。

我的总体计划的第1部分是限制在设计阶段所花费的时间。为了实现这一点，我在编写代码之前要对设计进行仔细考虑。但是，我相信在这个阶段进行一些试验是必要的，它们可以帮助我想出这个问题的解决方案。我的总体计划也允许创建2个项目：一个是粗略的原型，另一个是经过优化的最终解决方案。因此，我允许自己在真正的设计工作开始之前为那个原型编写代码，但不允许在最终解决方案中编写代码，除非我相信自己的设计已经完成。这并不能保证我可以对第2个项目的设计完全满意，但它为实现这个目标提供了最好的机会。

现在是时候对问题进行划分了。在前面各章中，我们曾列出完成一个问题所需要的所

有子任务，因此在这里我也将列出一个子任务列表。但是，现在完成这个任务还是有点困难的，因为我不知道这个程序将怎样进行作弊。我需要对这个问题进行进一步研究。

8.2.2 寻找作弊方法

绞刑者程序的作弊是一个非常具体的问题，我相信通过搜索常规的组件来源是无法找到任何帮助的。不可能存在像 NefariousStrategy（违法乱纪策略）这样的模式。现在，关于怎样进行作弊，我只有一个非常模糊的思路。我考虑列可以在一开始选择一个难点的单词，只要玩家 2 选择了这个单词中没有出现的字母就一直沿用它。但是，当玩家 2 猜中了这个单词中的一个字母时，我可以换个单词，选择一个所包含的字母到目前为止还没有被猜过的单词。换句话说，我将尽可能地否认玩家 2 的猜测。这是一种思路，但光有这个思路是不够的，我还需要一些可以实现的东西。

为了验证自己的思路，我打算在纸上模拟玩家 1，对一个单词列表进行操作。为了简单起见，假设玩家 2 所设定的单词长度为 3 个字母，并且表 8.1 列出了我所知道的所有 3 字母的单词。我假设自己所选择的第一个“问题单词”是列表中的第 1 个单词 bat。如果玩家 2 所猜的字母不是 b、a 或 t，我就说“错”，然后朝胜利又迈进了一步。如果玩家 2 猜中了这个单词中的任意一个字母，我就挑选另一个单词，只要它不包含玩家 2 之前所猜测的任何字母。

但是，看了这个表格之后，我无法保证这种策略是最好的。在有些情况下，它可能是合理的。假设玩家 2 猜了 b。这个表格中的其他单词都不包含 b，因此我可以切换到其他任何单词。这意味着我最大限度地降低了风险，我只是从单词列表中消除了 1 个可能的单词。但是，如果玩家 2 接着猜了 a 会怎么样呢？如果我回答“错”，我就必须排除所有包含了 a 的单词，这样就只剩下表 8.1 第 2 列的那 3 个单词可以供我选择。如果我决定承认字母 a 是正确的，那么我还剩下 5 个单词可以选择，如第 3 列所示。但是，注意这种扩展选择只有当 5 个单词中的 a 都处于同一位置时才成立。一旦声明某次猜测是正确的，我必须指出这个字母出现在单词中的位置。剩下的、可以用来应付未来猜测的单词越多，我对游戏获胜的信心就越足。

表 8.1 示例单词列表

所有单词	不带 a 的单词	带 a 的单词
bat	dot	bat
car	pit	car
dot	top	eat
eat		saw
pit		tap
saw		
tap		
top		

比外，即使我设法避免在游戏中早早就暴露字母，但还是要预计玩家 2 最终是否会做出正确的猜测。例如，玩家 2 可以从所有的元音字母开始猜测。因此，当一个字母被暴露时我必须决定该怎么应对。根据这个示例列表的试验，我必须要找到这个字母最常出现的位置。从这个角度出发，我认识到用这种方法进行作弊是错误的。我不应该事先挑选一个单词，即使是临时的，只要追踪所有可以选择的有可能性的单词就可以了。

有了这个思路之后，我可以用一种不同的方式来定义作弊：使候选单词列表中的单词尽可能多。对于玩家 2 所做出的每次猜测，这个程序必须要做出一个决定。它应该表示猜中还是猜错呢？如果表示猜中，被猜中的字母出现在哪个位置？我会让程序维护一个不断减少的候选单词列表，并在每次猜测之后，所做出的决定就是使这个列表保留尽可能多的单词。

8.2.3 绞型者作弊所需要的操作

现在我已经对问题有了足够的理解，可以创建自己的子任务列表了。对于同等规模的问题，在早期就列出这样一个清单可以有很好的机会能够确定一些操作。这是毫无疑问的，因为我在总体计划中已经预料到自己不会第一次就创建出一个完美的设计。

存储并维护一个单词列表

这个程序必须维护一个合法的英语单词列表。因此，这个程序必须从一个文件中读取一个单词列表并在内部以某种格式存储它们。当程序开始作弊后，随着游戏地不断进行，这个列表中的单词将不断减少。

创建一个特定长度的单词子列表

假设意图是维护一个候选问题单词的列表，那么我必须从一个由玩家 2 所指定长度的单词列表开始游戏。

追踪被选择的字母

这个程序需要记住哪些字母已经被猜过，其中有哪些是不正确的、哪些看上去是正确的以及它们在问题单词中所出现的位置。

没有出现某个字母的单词计数

为了便于进行作弊，我需要知道列表中有多少单词并不包含最近被猜测的字母。记住，程序将决定这个最近被猜测的字母是否出现在问题单词中，其目的是使候选单词列表保留尽可能多的单词。

根据字母和位置确定最大数量的单词

这看上去是难度最大的操作。我们假设玩家 2 刚刚猜了字母 d 并且当前游戏所设定的问题单词的长度为 3。当前的候选单词列表中可能一共有 10 个单词包含了 d，但这并不重要，关键在于程序必须说明这个字母出现在问题字母中的哪个位置。我们把字母在单词中的位置称为模式。因此，d??就是一个三字母模式，它的第 1 个字母为 d，另两个字母可以是 d 之外的任意字母。如表 8.2 所示。假设第 1 列的单词列表包含了程序所知的所有含 d 的三字母单词。另外几列按照该模式对这个列表进行分解。出现次数最多的模式是??d，一共有 17 个单词。17 这个数量将与候选列表中不包含 d 的单词数量进行比较，以确定程序应该表示猜中还是猜错。

表 8.2 三字母单词

所有单词	?dd	??d	d??	d?d
add	add	aid	day	did
aid	odd	and	die	
and		bad	doe	
doe		bed	dog	
dog		bid	dry	
dry		end	due	
due		fed		
did		had		
die		hid		
doe		kid		
dog		led		
dry		mad		
due		mod		
end		old		
fed		red		
had		rid		
hid		sad		
kid				
led				
mad				
mod				
odd				
old				
red				
rid				
sad				

创建一个与模式匹配的单词子列表

当程序声明玩家 2 猜中一个字母后，它将创建一个新的候选单词列表，只包含与所选

择的字母模式匹配的单词。在前一个例子中，如果我们声明 d 为猜中，那么表 8.2 的第 3 列将成为新的候选单词列表。

在游戏结束之前一直进行

所有操作就绪之后，我需要编写代码，把所有的操作集成在一起并实际运行游戏。这个程序应该反复请求玩家 2（用户）进行猜测、确定拒绝或接受这次猜测所剩下的候选单词列表哪个更长，并相应地削减这个单词列表，然后根据已揭晓的所有正确猜中的字母以及此前所猜测的所有字母，显示最终的问题单词。这个过程将一直持续，直到游戏结束，即其中一位玩家获胜。因此，我还需要制订游戏在什么条件下可以结束。

8.2.4 初始设计

尽管前面列出的所需操作列表看上去是比较粗略的，但设计决策却是根据这些操作做出的。考虑“创建一个与模式匹配的单词子列表”这个操作。这个操作将出现在我的解决方案中，至少将出现在它的初始版本中。但是，严格地说，它根本不是必需的操作。“创建一个特定长度的单词子列表”操作同样不是。我并不是为了维护一个不断变小的候选问题单词列表，而是如何在整个游戏过程中保留一开始时的问题单词主列表。但是，这将导致其他操作变得复杂。“没有出现某个字母的单词计数”操作不能仅仅迭代候选问题单词列表，并对不包含指定字母的所有单词进行计数。由于将对主列表进行搜索，因此它还必须检查每个单词的长度，并判断这个单词是否与到目前为止问题单词所暴露的字母匹配。我觉得自己所选择的路径在总体上变得更容易了，但必须意识到即使是早期的选择也会影响最终的设计。因此，除了在开始阶段把问题分解为子任务之外，我还必须做出其他决定。

怎样存储单词列表

这个程序的关键数据结构是单词列表，程序将在游戏期间不断缩减这个列表。在选择数据结构时，我进行了下面这些考虑。首先，我相信并不需要对列表中的单词进行随机访问，而是需要把这个列表从头至尾作为整体进行处理。其次，我并不知道这个列表的初始长度。再次，我需要经常对这个列表进行削减。最后，这个程序经常会使用标准 `string` 类的方法。综合上面这些因素，我决定这个结构的初始选择是标准模板库的 `list` 类，并用 `string` 作为它所包含的数据项的类型。

怎样追踪被猜的字母

被选择的字母在概念上是一个 set。也就是说，一个字母或者被选择，或者没有被选择，并且每个字母被选择的次数不超过 1 次。因此从本质上说，这个问题就是字母表中的一个

特定字母是否为一个“被选择的”set 的一个成员。因此，我打算用一个长度为 26 的 bool 类型的数组来表示被选择的字母。如果这个数组被命名为 guessedLetters，并且 a 在游戏进行过程中已经被猜过，那么 guessedLetters[0] 就是 true，否则就为 false。guessedLetters[1] 表示 b，接下来以此类推。我将使用本书一直使用的范围转换技巧在小写字母 a 和它在数组中的对应位置之间进行转换。如果 letter 是个表示小写字母的字符，那么 guessedLetters[letter - 'a'] 就是它在数组中的对应位置。

怎样存储位置模式

我将要实现的一个操作“创建一个与模式匹配的单词子列表”将使用单词中某个字母的位置模式。这个模式是由另一个操作“根据字母和位置确定最大数量的单词”所产生的。因此，我应该用什么格式表示这种数据呢？模式是一系列的表示一个特定字母所出现位置的数字。存储这些数字的方法有很多种，但我为了保持简单，使用另一个 list，数据项的类型为 int。

是不是应该编写一个类

由于我用 C++编写这个程序，因此可以自己决定是否使用面向对象编程。我首先想到的是上面列表中的许多操作可以很自然地放在一个称为 wordList 的类中，再添加根据指定的标准（即长度和模式）删除单词的方法。但是，由于想尽量避免做出以后可能会被撤销的设计决定，因此我打算在第一个粗略的试验程序中完全采用过程性编程。一旦我已经解决了这个程序的困难部分并实现了操作列表中的所有操作，就可以更好地决定是否在最终的版本中采用面向对象编程。

8.2.5 开始编写化码

现在可以开始做有趣的事了，我打开自己的开发环境并投入工作。这个程序将要使用标准库中的一些类，为了清晰起见，我首先列出了所有头文件和相关的 using 指令：

```
#include <iostream>
using std::cin;
using std::cout;
using std::ios;
#include <fstream>
using std::ifstream;
#include <string>
using std::string;
#include <list>
using std::list;
using std::iterator;
#include <cstring>
```

现在，我准备为自己的操作列表中的操作编写代码了。在某种程度上，我可以按照任意顺序为这些操作编写代码，但首先打算编写的函数是从一个包含单词的普通文本文件中把单词读取到自己所选择的 list<string>结构中。在这个时候，我意识到需要找到一个现成的单词文件，而不用自己输入这些单词。幸运的是，用 Google 搜索“word list”显示了一些提供普通文本格式的英语单词（每行显示一个单词）的网站。我已经掌握了怎样在 C++中读取文本文件，但即使自己并不熟悉这个操作，也可以编写一个小型的测试程序来练习这个技巧然后再用于绞型者作弊程序。我将在本章的后面内容中讨论这种做法。

找到了文件之后，我可以编写这个函数：

```
list<string> readWordFile(char * filename) {
   list<string> wordList;
 ❶ifstream wordFile(filename, ios::in);
 ❷if (wordFile == NULL) {
      cout << "File open failed. \n";
      return wordList;
   }
   char currentWord[30];
 ❸while (wordFile >> currentWord) {
    ❹if (strchr(currentWord, '\'') == 0) {
        string temp(currentWord);
        wordList.push_back(temp);
     }
   }
   return wordList;
}
```

这个函数非常简单，因此我只进行一些简要的说明。ifstream 对象❶表示一个输入流，它的工作方式就像 cin 一样，区别在于它是从一个文件中而不是从标准输入流中进行读取。如果它的构造函数无法打开这个文件（通常意味着无法找到这个文件），那么这个对象就为 NULL，这是我明确检查的一种情况❷。如果这个文件存在，就在一个循环中对它进行处理❸。这个循环每次把这个文本中的一行文本读取到一个字符数组中，并把这个数组转换为一个 string 对象，然后再把它添加到一个 list 对象中。我最终所使用的英语单词文件包含了带撇号的单词，它们对于这个游戏是不合法的，因此将明确排除它们❹。

接着，我编写一个函数，显示这个 list<string>中的所有单词。这并不是必需的操作，我不会在游戏中使用这个函数（毕竟，它只能帮助被欺骗的玩家 2），但它很适合测试我的 readWordFile 函数是否正确地完成了任务：

```
void displayList(❶const list<string> & wordList) {
 ❷list<string>::const_iterator iter;
   iter = wordList.begin();
   while (iter != wordList.end()) {
      cout << ❸iter->c_str() << "\n";
      iter++;
   }
}
```

它在本质上与前一章所介绍的列表遍历代码是相同的。注意，我把它的参数声明为常量引用❶。由于这个 list 对象在一开始时可能相当庞大，传递引用参数可以减少函数调用的开销，而传递值参数就必须复制整个 list。把这个引用参数声明为常量参数可以表示这个函数将不会修改 list，这就使它的代码更容易读懂。常量 list 需要使用常量迭代器❷。cout 流无法输出 string 对象，因此这个方法使用 c_str() 函数生成对应的以 null 字符结尾的字符数组❸。

我使用相同的结构编写了一个函数，以完成对 list 对象中不包含某个指定字母的单词进行计数：

```
int countWordsWithoutLetter(const list<string> & wordList, char letter) {
   list<string>::const_iterator iter;
   int count = 0;
   iter = wordList.begin();
   while (iter != wordList.end()) {
    ❶if (iter->find(letter) == string::npos) {
        count++;
      }
      iter++;
   }
   return count;
}
```

正如我们所看到的那样，这是一个相同的基本遍历循环。在循环内部，我调用了 string 类的 find 方法❶，它返回它的 char 参数在这个 string 对象中的位置。如果未找到这个字符，就返回一个特殊值 npos。

我使用相同的基本结构编写了一个函数，以完成从单词列表中删除所有与指定长度不匹配的单词：

```
void removeWordsOfWrongLength(list<string> & wordList,
                              int acceptableLength)
{
   list<string>::iterator iter;
   iter = wordList.begin();
   while (iter != wordList.end()) {
```

```
      if (iter->length() != acceptableLength) {
       ❶iter = wordList.erase(iter);
      } else {
       ❷iter++;
      }
   }
}
```

这个函数是一个非常好的例子，说明了我们所编写的每个程序都为自己提供了一个很机的机会以加深对程序工作方式的理解。这个函数对我来说很容易编写，因为通过自己以前所编写的程序理解了“幕后”所发生的事情。这个函数使用了与前面的函数相同的遍历代码，但循环内部的代码就比较有趣。erase()方法从一个 list 对象中删除一个由迭代器所指定的数据项。但是，根据我们在第 7 章实现链表类的迭代器模式的经验，我们知道迭代器可以肯定就是指针。根据我们在第 4 章的指针操作经验，我们知道当一个指针是野引用、指向已经被删除的东西时，它没什么用处并且往往非常危险。因此，我知道需要在这个操作之后为 iter 赋一个合法的值。幸运的是，erase()的设计者已经预料到了这个问题并通过这个方法返回一个新的迭代器，指向被删除的元素之后紧随的那个元素。因此，我可以把这个值赋值给 iter❶。另外，注意我只有在没有从 list 中删除当前字符串时才明确把 iter 推进一步❷，因为用 erase()的返回值进行赋值时已经把这个迭代器向前推进了一步，我不想跳过任何数据项。

现在开始进入困难的部分：在剩余的单词列表中寻找一个指定字母的最常见模式。这是使用分治法技巧的另一个机会，我知道这个操作的其中一个子任务是确定一个特定的单词是否与一个特定的模式匹配。记住，模式就是 list<int>对象，其中每个 int 值表示字母出现在单词中的位置。因此，为了让一个单词匹配一个模式，这个字母必须出现在单词中的指定位置，而不能出现在单词中的其他位置。有了这个思路之后，我就可以通过遍历一个字符串来测试它是否匹配了。对于这个字符串中的每个位置，如果出现了指定的字母，我就确定这个位置是在模式中。如果出现了其他字母，我就确定这个位置不是在模式中。

为了简单起见，我首先编写一个单独的函数，以检查一个特定的位置编号是否出现在一个模式中：

```
bool numberInPattern(const list<int> & pattern, int number) {
   list<int>::const_iterator iter;
   iter = pattern.begin();
   while (iter != pattern.end()) {
      if (*iter == number) {
         return true;
```

```
            return true;
         }
         iter++;
      }
      return false;
   }
```

熟悉了前几个函数之后，这段代码显得相当简单。我简单地对 list 进行遍历，搜索 number。如果找到就返回 true，如果到达 list 的尾部还没找到就返回 false。现在，我可以实现通用的模式匹配测试了：

```
bool matchesPattern(string word, char letter, list<int> pattern) {
   for (int i = 0; i < word.length(); i++) {
      if (word[i] == letter) {
         if (!numberInPattern(pattern, i)) {
            return false;
         }
      } else {
         if (numberInPattern(pattern, i)) {
            return false;
         }
      }
   }
   return true;
}
```

正如我们所看到的那样，这个函数遵循了之前所制订的计划。对于这个字符串中的每个字符，如果它与 letter 匹配，就确认当前位置是在这个模式中；如果这个字符并不与 letter 匹配，就确认这个位置并不在这个模式中。如果一个位置并不与这个模式匹配，这个单词就被拒绝；否则，遍历到这个单词尾部的时候，这个单词就被接受。

此时，我感觉到如果 list 中的每个单词都包含了指定的字母，寻找最常见模式的任务就会容易很多。因此，我编写了一个简单的函数，以去掉那些不包含这个字母的单词：

```
void removeWordsWithoutLetter(list<string> & wordList,
                              char requiredLetter) {
   list<string>::const_iterator iter;
   iter = wordList.begin();
   while (iter != wordList.end()) {
      if (iter->find(requiredLetter) == string::npos) {
         iter = wordList.erase(iter);
      } else {
         iter++;
      }
   }
}
```

这段代码只是前几个函数所使用的思路的组合。既然我已经想到了这个函数，就需要一个相反的函数去掉那些包含了指定字母的单词。我将使用这个函数，当程序表示最近一次猜测为错时对候选单词列表进行削减。

```
void removeWordsWithLetter(list<string> & wordList, char forbiddenLetter) {
   list<string>::const_iterator iter;
   iter = wordList.begin();
   while (iter != wordList.end()) {
      if (iter->find(forbiddenLetter) != string::npos) {
         iter = wordList.erase(iter);
      } else {
         iter++;
      }
   }
}
```

现在，我准备为特定的字母寻找单词列表中最常见的模式了。我思考了一些方法并挑选了自己认为最容易实现的一种。首先，我调用上面的函数以删除所有不包含指定字母的单词。接着，我提取列表中的第1个单词，确定它的模式，并对列表中具有相同模式的其他单词进行计数。当我进行计数时，它们都将从列表中被删除。然后，对现在位于列表头部的那个单词执行相同的处理，直到这个列表为空。其结果如下所示：

```
void mostFreqPatternByLetter(❶list<string> wordList, char letter,
                             ❷list<int> & maxPattern,
                             ❸int & maxPatternCount) {
 ❹removeWordsWithoutLetter(wordList, letter);
  list<string>::iterator iter;
  maxPatternCount = 0;
 ❺while (wordList.size() > 0) {
     iter = wordList.begin();
     list<int> currentPattern;
   ❻for (int i = 0; i < iter->length(); i++) {
        if ((*iter)[i] == letter) {
           currentPattern.push_back(i);
        }
     }
     int currentPatternCount = 1;
     iter = wordList.erase(iter);
   ❼while (iter != wordList.end()) {
        if (matchesPattern(*iter, letter, currentPattern)) {
           currentPatternCount++;
           iter = wordList.erase(iter);
        } else {
           iter++;
        }
     }
```

```
        ❽if (currentPatternCount > maxPatternCount) {
            maxPatternCount = currentPatternCount;
            maxPattern = currentPattern;
          }
          currentPattern.clear();
        }
    }
```

这个列表之所以作为值参数传递的❶，是因为这个函数在处理过程中将把这个列表清空，而我并不想影响这个函数的调用代码实际所传递的参数。注意 maxPattern❷和 maxPatternCount❸都只是作为输出参数。它们将把最常出现的模式和出现次数传递给这个函数的调用代码。我删除了不包含 letter 的所有单词❹。接着，进入这个函数的主循环，它在列表不为空时将一直继续❺。循环内部的代码分为三个主要部分。首先，一个 for 循环构建了列表第 1 个单词的模式❻。接着，一个 while 循环对列表中与这个模式匹配的单词进行计数❼。最后，我采用第 3 章所讨论的“山丘之王”策略，观察这个计数是否大于到目前为止的最大计数值❽。

我所需要的最后一个功能函数将显示到目前为止猜测的所有字母。读者应该还记得我把它们存储在一个包含 26 个 bool 值的数组中：

```
void displayGuessedLetters(bool letters[26]) {
   cout << "Letters guessed: ";
   for (int i = 0; i < 26; i++) {
      if (letters[i]) cout << ❶(char)('a' + i) << " ";
   }
   cout << "\n";
}
```

注意，我把一个范围的基本值（在这个例子中就是字符 a）加上另一个范围的一个值❶，这个技巧是在第 2 章中首先采用的。

现在，我已经完成了所有的关键子任务，可以尝试解决整个问题了。但是，还有很多函数未经过完全的测试，我想尽快对它们完成测试。因此，我并不打算在一个步骤中处理剩余的所有问题，而是打算对问题进行削减。为此，我打算把有些变量（如问题单词的长度）作为常量。

由于我最终将会抛弃这个版本的程序，因此可以毫无顾虑地把整个游戏的逻辑放在 main 函数中。但是，由于结果非常长，因此我打算分阶段显示这些代码。

```
int main () {
 ❶list<string> wordList = readWordFile("wordlist.txt");
   const int wordLength = 8;
   const int maxMisses = 9;
 ❷int misses = 0;
 ❸int discoveredLetterCount = 0;
 ❹removeWordsOfWrongLength(wordList, wordLength);
 ❺char revealedWord[wordLength + 1] = "********";
 ❻bool guessedLetters[26];
   for (int i = 0; i < 26; i++) guessedLetters[i] = false;
 ❼char nextLetter;
   cout << "Word so far: " << revealedWord << "\n";
```

第1部分的代码设置了玩这个游戏时所需要的常量和变量，这段代码大部分看上去都非常容易理解。单词列表是根据一个文件创建的❶，然后削减为指定的单词长度，在此例中为常量值8❹。变量 misses❷用于存储玩家2猜错的次数，而变量 discoveredLetterCount❸用于追踪这个单词所暴露的位置个数（因此，如果d出现了2次，猜d就把这个值增加了2）。变量 revealedWord 用于存储玩家2当前已知的问题单词，用星号表示到目前为止还没有被猜中的字母❺。bool 类型的数组 guessedLetters❻用于追踪到目前为止所猜测的特定字母，用一个循环把它的所有值都设置为 false。最后，变量 nextLetter❼用于存储玩家2的当前猜测。我输出 revealedWord 初始值，然后进入主要的游戏循环。

```
❶ while (discoveredLetterCount < wordLength && misses < maxMisses) {
    cout << "Letter to guess: ";
    cin >> nextLetter;
  ❷guessedLetters[nextLetter - 'a'] = true;
  ❸int missingCount = countWordsWithoutLetter(wordList, nextLetter);
    list<int> nextPattern;
    int nextPatternCount;
  ❹mostFreqPatternByLetter(wordList, nextLetter, nextPattern, nextPatternCount);
    if (missingCount > nextPatternCount) {
     ❺removeWordsWithLetter(wordList, nextLetter);
      misses++;
    } else {
     ❻list<int>::iterator iter = nextPattern.begin();
      while (iter != nextPattern.end()) {
        discoveredLetterCount++;
        revealedWord[*iter] = nextLetter;
        iter++;
      }
      wordList = reduceByPattern(wordList, nextLetter, nextPattern);
    }
    cout << "Word so far: " << revealedWord << "\n";
    displayGuessedLetters(guessedLetters);
}
```

出现了以下两种情况之一就可以结束游戏。其一，玩家 2 发现了单词中的所有字符，使 discoveredLetterCount 达到了 wordLength 的值；其二，玩家 2 的猜错次数达到了 maxMisses 值。因此，只要这两个条件都没有被满足，这个循环就会一直持续❶。在循环内部，从用户那里读取了下一次猜测字母之后，guessedLetters 数组中对应的位置就会被更新❷。然后，作弊就开始了。程序调用 countWordsWithoutLetter❸，确定如果把这次猜测声明为错误时单词列表中还剩下多少个候选单词，并调用 mostFreqPatternByLetter❹函数确定如果承认这次猜中最多还能剩下多少个候选单词。如果前者更大，就删除含有被猜测字母的单词，并且把猜错数加 1❺。如果后者更大，就采用 mostFreqPatternByLetter 函数所提供的模式并更新 revealedWord 值，同时从单词列表中删除所有与这个模式不匹配的单词❻。

```
    if (misses == maxMisses) {
        cout << "Sorry. You lost. The word I was thinking of was '";
        cout << ❶(wordList.cbegin())->c_str() << "'.\n";
    } else {
        cout << "Great job. You win. Word was '" << revealedWord << "'.\n";
    }
    return 0;
}
```

代码的剩余部分被我称为“验尸循环”，循环之后的操作是由“杀死”循环的条件所决定的。在这里，要么是我们的程序成功地通过作弊获得胜利，要么是玩家 2 克服了所有的困难迫使程序显示完整的单词。注意，当程序获胜时，列表中至少还保留一个单词❶，因此我只显示第 1 个单词，并声称这就是我所挑选的单词。更邪恶的程序可能会在剩余的单词中随机地挑选一个，以减少对手发现作弊的可能性。

8.2.6 对初始结果的分析

我已经把所有代码集中在一起并对它进行了测试，它能够完成任务，但很显然还需要很多改进。除了所有设计方面的考虑之外，这个程序还缺少很多功能：它并不允许用户指定问题单词的长度或允许猜错的次数；不检查被猜测的字母之前是否被猜过；更有甚者，它甚至没有检查输入的字符是不是小写字母；它还缺少很多界面友好性，例如没有告诉用户目前还允许猜错几次。另外，我觉得如果程序提供再玩一次的功能，而不是生硬地迫使用户重新运行这个程序会更好一些。

至于设计方面，当我思考程序的最终版本时，打算严肃地考虑采用面向对象设计。单词列表类看上去像是一个很自然的选择，而上面的 main 函数实在是过于庞大了。我喜欢采用模块

化的、易于维护的设计。我觉得 main 函数应该尽量短小精悍，它的作用就是指挥各个子程序完成真正的工作。因此，上面的 main 函数需要分解为几个函数。另外，我的一些初始设计选择可能需要重新加以考虑。例如，站在事后诸葛的角度，以 `list<int>`的形式存储模式似乎显得有些臃肿。也许我可以考虑采用一个 `bool` 类型的数组，就像 `guessedLetters` 一样。

也许我应该寻找另一种结构。现在我还需要回过头来观察在解决这个问题时是否有机会学习新的技巧。我在思考是否存在自己没有考虑到的特殊数据结构能够帮忙。尽管我最终还是决定采用原先的选择，但仍然可以从这种思考中学到很多东西。

尽管所有这些决定仍然是若隐若现的，但我觉得正沿着自己的道路前进。让一个工作程序满足问题的本质需求是一个非常重要的步骤。我可以非常轻松地在原始版本中对不同的设计思路进行试验，自己的信心来源就是已经拥有了一个解决方案，只不过现在的目标是寻找一个更好的解决方案而已。

创建还原点

Microsoft Windows 操作系统在安装或更改系统组件之前会创建一个还原点。还原点包含了关键文件的备份拷贝，例如注册表。如果一次安装或更新导致了严重的问题，可以通过复制这些文件有效地“回滚”或撤消到还原点。

我强烈建议在自己的源代码中采用与还原点相同的方法。当我们已经有了一个预期在将来会进行修改的工作程序之后，应该创建整个项目的一份拷贝，然后只对这份拷贝进行修改。这种做法非常省力，如果以后发觉自己的修改不合适，就可以节省大量的时间。程序员很容易陷入这样的思维陷阱：“我已经完成过这件事件，因此再完成一次肯定没问题”。通常情况下这是正确的，但是在“知道自己可以重新做某件事情”和“直接拿来旧的源代码供直接参考”之间还是存在很大的差别的。

我们还可以使用版本控制软件，它实现了复制和存储项目文件的自动化。版本控制软件所实现的功能不仅仅是“还原点”功能。例如，它还允许多个程序员独立地对同一批文件进行操作。虽然对这类软件的讨论超出了本书的范围，但是作为程序员，还是应该对此有所了解。

8.2.7 解决问题的艺术

目前为止，读者是否已经发现了我在这个解决方案中采取的所有问题解决技巧？我为解决这个问题制订了一个计划。和往常一样，这是所有问题解决技巧中最为关键的一个。

我决定首先完成第一个版本的解决方案，采用自己非常熟悉的数据结构即数组和 `list` 类。我简化了程序的功能，这样更容易编写这个粗略而可行的版本，并以最快的速度对代码进行测试。我把问题划分为不同的操作，并把每个操作实现为一个不同的函数，这样就可以对程序的不同片段进行独立地操作。当还不清楚怎样进行作弊时，我进行了试验，把“作弊”重新表述为“最大限度地保留候选单词列表的长度”，这对我而言是一个可以实现的具体概念。在实现所有操作的特定细节方面，我采用了本书中所讨论的各种问题解决技巧。

我还成功地避免了遭遇挫折，读者也要对自己充满信心。

在继续讨论之前，读者必须清楚地理解我在这个问题的解决过程中所采取的步骤。读者在解决这个问题时并不一定要采取相同的步骤。上面所显示的代码不见得是这个问题的最佳解决方案，也不一定比读者所想出来的方法更合适。我希望这个例子能说明不管问题的规模如何，它们都可以用本书所使用的各种基本技巧进行解决，只不过有时需要采取一些变化。如果读者所面临的问题规模要比这个例子复杂 1 倍甚至 10 倍，它可能对读者的毅力是很大的挑战，但是读者最终还是有能力解决它。

8.3 学习新的编程技能

还有一个话题需要讨论。在精通本书的问题解决技巧之后，如果我们想把程序员作为自己毕生的事业，还需要走出关键的一步。但是，就像大部分职业一样，这是一条没有终点的道路，因此我们必须设法完善自己，成为更优秀的程序员。和编程中的每个任务一样，我们也应该为自己学习新技巧和新技术制订计划，而不是东一榔头西一锤子式地攫取新知识。

在本节中，我们将讨论读者可能想要获取新技巧的一些领域，并讨论每个领域的一些系统性的学习方法。所有领域的一个共同特点是必须把想要学习的东西运用在实践中。这也是本书的每一章都以习题结尾的原因，读者应该已经完成所有的习题了吧？阅读新的编程思路是学会它们的关键第一步，但也仅仅是第一步。为了达到在现实世界的问题的解决方案中熟练运用一种新技巧的程度，首先应该在一个规模较小的问题中试验这个新技巧。记住，我们的基本问题解决技巧之一就是对问题进行削减，以使自己所处理的每个状态只有一个实质性的元素。我们不需要在学习新技巧的同时解决一个将成为解决方案核心的重要问题，因此这样一来，自己的注意力会被两个困难问题所分散。

8.3.1 新语言

我觉得 C++在编写代码方面是一种优秀的语言，并且在第 1 章已经解释了为什么它还

是一种非常适合学习的语言。没有一种语言在所有方面都比其他语言更为出色，因此优秀的程序员应该学习几种不同的语言。

花时间学习

尽可能在使用一种新语言编写代码之前，留出时间对它进行研究。如果读者用一种以前从未使用过的语言解决一个不是很简单的问题，就违背了一个重要的问题解决规则：避免受挫。为自己设置任务学习一种语言，在完成这个任务之后才能用这种语言完成“真正的”程序。

当然在现实世界中，我们有时候无法完全控制分配给自己的项目。在任何时刻，我们都可能接到请求用一种特定的语言编写一个程序，并且这个请求的时间期限非常紧张，在处理实际问题之前没有多少时间供我们悠闲地研究这种语言。防止出现这种情况的最好方法是在使用一种新语言之前就未雨绸缪，对它有所研究。我们应该对自己感兴趣的语言或者在以后的编程工作中很可能会用到的语言进行研究。这是从短期来看时间效率不高，但从长期来看能够收到丰厚回报的又一种情况。即使结果证明在不远的将来并不需要使用现在所研究的语言，但是对另一种语言的研究仍然能够提高自己所掌握语言的技能，因为它迫使你用新的不同方式进行思考，打破了旧思维方式，使你能够从新的视角审视自己的技巧和技术。我们可以把它看成是编程中的交叉培训。

从自己所知道的开始

当我们开始学习一种新的编程语言时，应该对它毫无了解。但是，如果它并不是我们所学习的第一种编程语言，那么我们实际上已经掌握了很多编程技巧。因此，学习一种新语言的一个非常好的方法就是理解怎样用这种新语言改写自己用另一种已经掌握的语言所编写的代码。

如前所述，我们需要通过实践来学习，而不能仅仅通过阅读。找到一些用其他语言所编写的程序，然后用这种新语言重新编写它们。对单独的语言要素（如控制语句、类、其他数据结构等）进行系统性地研究，目标是尽可能地把自己在其他语言上的知识和技能转移到这种新语言上。

研究区别所在

下一个步骤是研究新语言与已掌握语言的区别所在。虽然两种高级语言可能具有广泛的相似之处，但它们之间肯定存在不同的地方，否则也没有必要选择新语言来完成某个项目了。同样，我们要通过学习来完成这个目标。例如，仅仅通过阅读了解“一种语言的多

选择语句允许它的条件为某个范围的值（C++的 switch 语句只能使用单独的值）”的效果远远不如在实际编写代码时使用这个功能编写有意义的代码。

很显然，这个步骤对于那些存在显著匹别、但又同等重要、并且具有共同祖先的语言是极为重要的，例如 C++、C#和 Java，它们都是 C 的面向对象的后代。语法上的相似性使我们很容易误以为自己对新语言理解甚深，实际上却并非如此。考虑下面的代码：

```
integerListClass numberList;
numberList.addInteger(15);
```

如果这两行是以 C++代码的形式出现在你的面前，你会认为第 1 行创建了 integerListClass 类的一个对象，第 2 行调用了这个对象的 addInteger 方法。如果这个类确实存在并且具有接受一个 int 参数的方法，这段代码完全没有任何问题。现在，我告诉你这段代码是用 Java 而不是 C++编写的。从语法上说，这两行代码并没有不合法的地方。但是，在 Java 中，光凭类对象的一个变量声明并不能创建这个对象，因为这个对象变量实际上是个引用。也就是说，它的行为与指针相似。为了在 Java 中执行相同的步骤，正确的代码应该是：

```
integerListClass numberList = new integerListClass;
numberList.addInteger(15);
```

我们可能很快理解了 Java 和 C++之间的这个特定区别，但许多其他区别可能非常微妙。如果没有花时间去发现它们，那么在新语言进行调试时可能会变得极为困难。当我们回顾自己用新语言所编写的代码时，内心深处对其他语言的固有思维可能会向自己反馈不正确的信息。

研究优秀的代码

在本书中，我的一个观点是：不应该通过直接拿来别人的代码并对它进行修改来学习编程。但是，有时候研究其他人的代码是非常重要的。虽然可以通过编写一系列的原创程序来学习一种新语言的技能，但是为了达到精通的程度，我们还需要参考其他精通这种语言的人所编写的优秀代码。

我们并不是想要“剽窃”这样的代码，也不打算借用这段代码解决一个特定的问题。反之，我们通过观察现有的代码，以发现这种语言的“最佳实践”。观察编程专家的代码，不仅要明白这位专家想要做什么，还要明白他为什么要这样做。如果代码附有注释，那就再好不过了。要区分某种做法仅仅是风格上的选择还是为了实现性能的优化。完成了这个步骤之后，就可以避免一个常见的陷阱。程序员们常常对一种新语言有所了解之后就匆

匆上手，结果他们所编写的代码非常脆弱，并没有利用这种语言的所有特征。例如，你是一位被要求用 Java 编写代码的 C++程序员，显然不能安于用 C++编写代码。反之，你要按照 Java 程序员所采用的方式编写真正的 Java 代码。

和其他所有事物一样，要把自己所学习的知识投入到实践中。拿出原先的代码并对它进行修改，以完成一些新的任务。然后把代码扔在一边，尝试重新编写它。目标是能够对这样的代码得心应手，足以回答其他程序员所提出的相关问题。

需要强调的是，这个步骤出现在其他步骤之后。在到达研究其他人所编写的新语言代码这个阶段之前，我们应该已经学习了这种新语言的语法，并把自己通过其他语言所学习的问题解决技巧应用于这种新语言。如果我们首先研究用新语言所编写的长长的程序例子，并对这些例子进行修改来缩短学习过程，很可能会限制自己的学习深度。

8.3.2 已经熟悉的语言的新技巧

当你认为自己“知道”了一种语言时，并不会意识着你已经了解了这种语言的所有知识。即使精通了这种语言的语法，总是会存在新的方法能够组合现有的语言特性来解决问题。这些新方法大多已经在前一章讲述“组件”时进行了讨论，我们在第 7 章讨论了怎样创建组件知识库。其中一个重要的因素就是努力。一旦已经掌握了用某种方法解决问题的能力，就容易依赖自己已经知道的方法而停止对新事物的追求。这就像一位棒球投手能够投出一种犀利的快球，但不知道怎样投其他类型的好球。虽然有些投球手仅靠一种投球姿势就实现了成功的职业生涯，但是从一位看守员成长为先发队员，投球手还需要知道更多。

为了使自己成为的最优秀程序员，需要想法设法掌握新知识和新技能，并把它们投入到实践中。寻求挑战并克服它们，对自己所选择的语言的程序员专家所完成的工作进行研究。

记住，需求是发明的源泉，要想办法寻找无法用自己当前的技能圆满解决的问题。有时候，可以通过修改自己已经解决的问题以增加新的挑战。例如，我们可能已经编写了一个在数据集非常小的情况下能够顺利工作的程序，但是当允许数据增长到非常庞大的规模时会发生什么情况呢？或者，如果我们编写了一个在本地硬盘存储数据的程序，但是想远程存储数据时该什么办呢？如果我们需要一个程序的多个执行能够并发地进行远程访问和更新数据，应该怎么办呢？从一个工作程序开始并添加新的功能，就可以把注意力集中在这个新的编程方向上。

8.3.3 新代码库

现代的编程语言与它们的核心代码库是分不开的。例如，当我们学习 C++的时候，不可避免地会学到有关标准模板库的东西。当我们研究 Java 的时候，也会了解标准 Java 类。但是，除了与语言捆绑在一起的库之外，我们还需要研究第三方的代码库。有些是通用的应用程序框架，例如 Microsoft 的.NET 框架，它可以在几种不同的高级语言中使用。在某些情况下，库是某个领域所特定的，像用于图形的 OpenGL。还有一些库是第三方的专利软件包的组成部分。

和学习新语言一样，我们不应该在需要使用一个新库的主要项目中学习这个新库。反之，应该在一个重要性为零的测试项目中独立地学习这个库的主要组件，然后再把它们用于真正的项目。我们应该给自己布置难度不断增长的问题。记住，我们的目标并不是为了完成这些问题，而是通过这个过程进行学习。因此在自己的程序中成功地应用了这个新库之后，就不再需要对解决方案进行完善，甚至不需要将它们完成。然后，这些程序可以作为以后工作的参考。当我们因为不记得该怎么办而烦恼时，例如不知道怎样在 OpenGL 中把一个 2D 画面叠加到一个 3D 场景中，最好的办法莫过于打开一个自己以前所编写的专门用来演示这种技巧的旧程序，因为它是为此量身定做的。

另外，在学习一种新语言时，一旦掌握了一个库的基本内容，还应该阅读专家使用这个库所编写的代码。大多数大型的库具有官方文档并没有描述的特质或禁忌，只有具有长期经验的程序员才有可能发现它们。事实上，为了尽快地掌握某些库，需要使用其他程序员所提供的框架。重要的是，我们对别人代码的依赖不能超过必要的限度，应该尽快进入重新创建自己最初所看到的代码这个阶段。我们会惊奇地发现通过重建其他人的现成代码这个过程能够学到大量的东西。我们可能在原先代码中看到一个库函数的调用，并理解传递给这个函数的参数产生了一个特定的结果。但是，当我们把这段代码放在一边，并重新生成属于自己的效果时，就会迫使自己研究这个函数的文档，了解这个参数可以接受的所有特定的值，以及为了取得所需要的结果而必须把它们设置为什么。

8.3.4 上课

作为一名长期的教育工作者，我觉得在本节结束时必须考虑上课这个问题。不管我们想要学习的是编程的哪个领域，总能找到可以为我们上课的人，无论是在传统的课堂上，还是在一些在线环境中。但是，上课只是学习的一种催化剂，而不是学习本身，尤其是在编程这个领域。不管编程老师多么学识渊博、多么富有激情，但是当实际学习新的编程技能时，我们所面对的是自己的计算机，而不是坐在课堂上面对老师。正如我在本书中反复强调的那样，必须把编程思路投入到实践中，这样才能保证自己

真正学会它们。

这并不是说上课就没有价值，事实上它们常常具有非常巨大的价值。编程中的有些概念在本质上是难以理解的或者容易混淆，如果我们请教一位善于答疑解惑的教师，可以节省大量的时间和精力。另外，通过上课还可以对我们的学习进行评估。如果我们的教师尽心负责，可以从他对我们的代码的审阅中学到很多东西，可以极大地提高学习效率。最后，课业的成功完成可以给当前或未来的员工提供一些证明，表明我们已经理解了所学习的主题（就算运气不佳，所遇到的教师能力有限或责任心不足，起码我们有过这样的经历）。

要记住编程教育是我们的责任，就算我们只上了一堂课时。当一门课程在期末结束时，它可以提供评级的框架，但这个框架并不会限制我们的学习。要把自己的上课时间看成是尽可能多地学习相关主题的好机会，而并不仅局限于课程大纲所列的目标。

8.4 结论

我非常喜欢回想自己的第一次编程经历。我编写了一个简短的、基于文本的弹球机模拟程序。虽然听上去好象不可思议，但我在那个时候还没有自己的计算机，谁能在 1976 年就拥有自己的计算机呢？但是，在我父亲的办公室里，有一台电传打字机终端，它在本质上是一台巨大的点阵式打印机，拥有像算盘一样的键盘，并通过声频调制解调器与本地大学的主机进行通信（拿起电话用手拨号，听到嘶嘶的电子声时，把电话听筒放在一个与终端相连的特殊支架上）。虽然我的弹球模拟程序非常原始和粗糙，但是当这个程序开始运行并且计算机按照自己的指令操作时，我便深深地着迷了。

在那天我的感觉，计算机就像一堆无限的乐高积木、建筑模型和林肯汽车模型，可以让我建造自己能够想象得到的任何东西。正是这种感觉激发了我对编程的热爱。当开发环境宣布顺利生成了程序并且我的指尖放在键盘上等待程序的执行时，自己总会感到非常兴奋，那是一种对成功的期待和对失败的预感。同时渴望看到自己努力的成果，不管是编写一个简单的测试项目还是对一个大型解决方案进行最后的优化，不管是创建一个漂亮的图形程序还是创建一个数据库应用程序的前端。

我希望你在编程中能够产生相似的感觉。即使你还在努力弄懂本书所讨论的一些技术要点，我仍希望你能够理解：只要编程能够让你感觉到兴奋，能够让你深陷其中，就没有你解决不了的问题。最重要的就是你愿意投入精力并沿着正确的方向前进，时间会

接管剩余的一切。

你现在是不是能够像程序员一样思考了呢？如果你已经完成各章最后的习题，就应该已经能够像程序员一样思考，并对自己的问题解决能力充满信心。如果你还没有解决很多习题，我会向你提出一个建议，并且你也能够猜到，那就是完成更多的习题。如果你跳过了前面的一些章节，不要从本章的习题开始，回到你跳过的地方，从那个地方开始学习。如果你不想完成更多的习题 ，因为你不喜欢编程，那我也无能为力了。

一旦我们能够像程序员一样思考，就要为自己的技能感到自豪。如果有人叫你码农而不是程序员，奚落你说一只受过良好训练的鸟也能啄出代码，你可以反驳说自己并不仅仅是编写代码，而是使用代码来解决问题。当你坐在面试桌前接受未来雇主或客户的面试时，你要相信不管自己所面试的工作需要什么，你都能够满足其要求。

8.5 习题

现在只剩下最后一组习题了。当然，它们比以前各章的习题难度更大，并且更为开放。

8.1 编写绞刑者作弊问题的一个完整实现程序，要比作者的解决方案更优秀。

8.2 对自己的绞刑者问题进行扩展，使用户可以选择成为玩家 1。用户仍然要选择单词中的字母数以及允许猜错的次数，但是由程序进行猜测。

8.3 用另一种语言重新编写自己的绞刑者程序，这种语言应该是读者目前知之甚少或者完全不了解的。

8.4 把自己的绞刑者程序以图形的形式实现，显示绞刑架和绞刑者。我们应该以程序员而不是艺术家的目光来考虑图形，因此不用担心艺术质量。但是，我们必须创建一个真正的图形程序。不要用 ASCII 文本来绘制绞刑者，因为这太容易了。我们可以考虑用 C++的 2D 图形库或者选择一种更加面向图形的不同平台，例如 Flash。使用图形绞刑者可能要求对猜错次数进行限制，但应该有一种方法提供至少一个范围内的选择。

8.5 自行设计：采用在绞刑者问题中所学习的技巧解决另一个完全不同的、涉及单词列表的问题，例如使用单词的其他游戏，像拼字游戏、拼写检查程序或自己想到的其他程序。

8.6 自行设计：寻找一个规模或难度肯定是自己当前的技能所无法解决的 C++编程问题，想办法解决它。

8.7 自行设计：寻找自己感兴趣但还没有在程序中使用过的一个库或 API。对这个库或 API 进行研究，然后在一个实用的程序中使用它。如果对通用的编程感兴趣，可以考虑 Microsoft .NET 库或者一种开放源代码的数据库类库。如果喜欢低层的图形，可以考虑 OpenGL 或 DirectX。如果喜欢编写游戏，可以考虑像 Ogre 这样的开放源代码的游戏引擎。思考自己所编写的程序的类型，找到一个适合的库，并对它进行研究。

8.8 自行设计：为一个新平台（对自己而言是不熟悉的）编写一个实用的程序，例如，移动编程或网络编程。